TEMA 3

HERÈNCIA

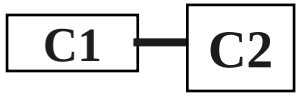
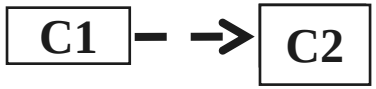
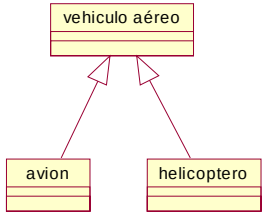
versió 3

Temporalització: 3 sessions

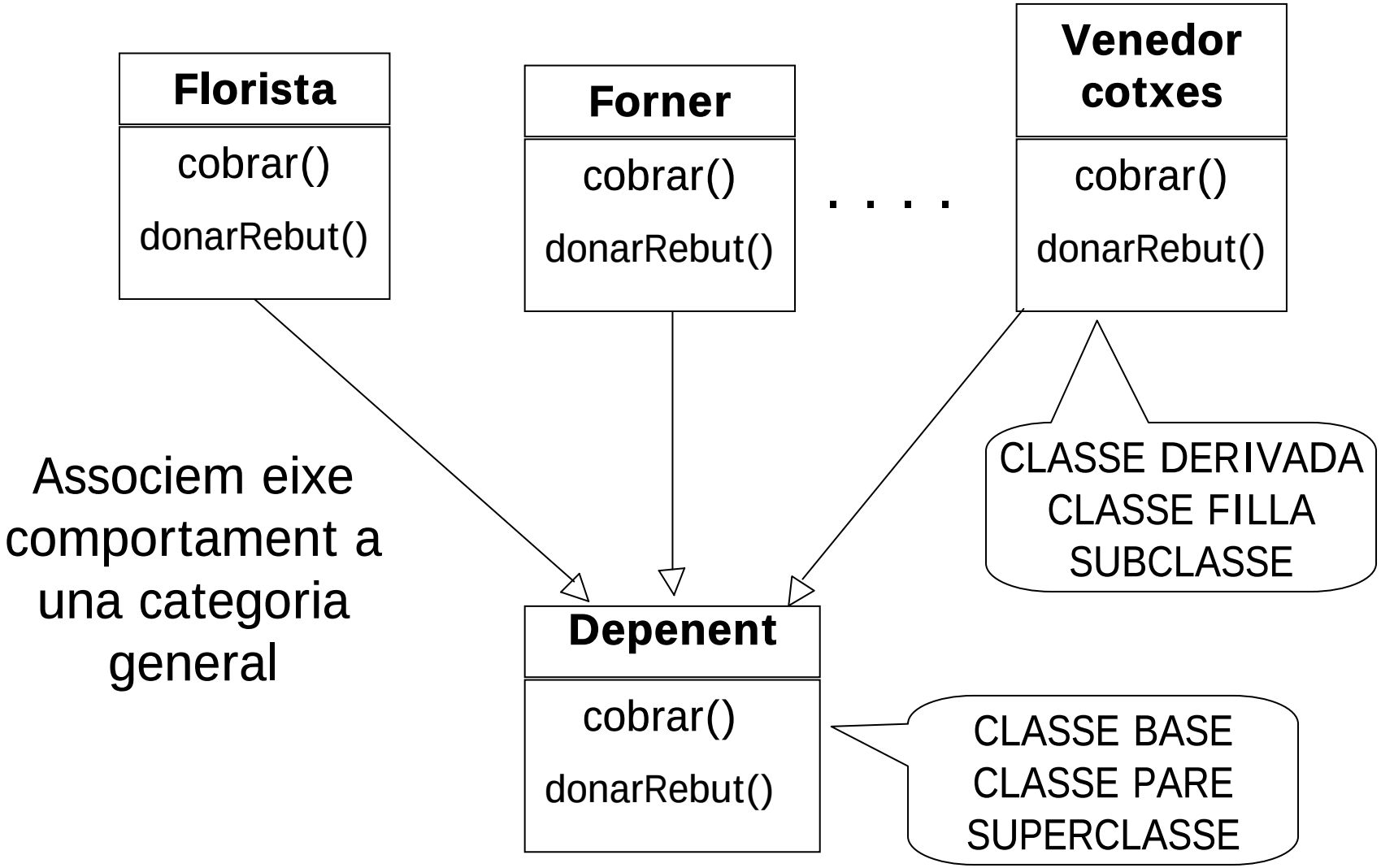
Tema 3. HERÈNCIA
Objectius

- Entendre el mecanisme d'abstracció de l'herència.
- Saber distingir entre jerarquies d'herència segures (bé definides) i insegures.
- Comprendre els costos de l'herència
- Reutilització de codi: Ser capaç de decidir quan usar herència i quan optar per composició.

Herència
Del tema anterior...

	Persistent	No persist.
Entre objectes	▪ **Asociació**  ▪ **Tot-Part** ▪ Agregació ▪ Composició 	▪ **Ús (depend)** 
Entre classes	▪ **Generalització** 	

HERÈNCIA
Motivació



Associem eixe comportament a una categoria general

HERèNCIA
Motivació

- L'herència és una de les característiques que més es destaca de la programació orientada a objectes, ja que gràcies a aquesta, és possible **especialitzar** o **estendre** la funcionalitat d'una classe, derivant d'ella noves classes
- L'herència és sempre **transitiva**, açò significa que una classe pot heretar característiques de superclasses que es troben molts nivells més amunt en la jerarquia d'herència.
  - Exemple: si la classe Gos és una subclasse de la classe Mamífer, i la classe Mamífer és una subclasse de la classe Animal, aleshores el Gos heretarà atributs tant de Mamífer com d'Animal.

HERÈNCIA
Test “ÉS-UN”

- La classe A s’ha de relacionar amb una relació d'herència amb la classe B si “A ÉS-UN B”. Si la frase sona bé, aleshores la situació d'herència és la més probable per eixe cas
  - Un ocell és un animal
  - Un gat és un mamífer
  - Un pastís de poma és un pastís
  - Una matriu d'enters és una matriu
  - Un cotxe és un vehicle

HERÈNCIA
Test “ÉS-UN”

- No obstant això, si la frase sona rara per una raó o altra, és molt probable que la relació d'herència no siga el més adequat. Vegem uns exemples:
  - Un ocell és un mamífer
  - Un pastís de poma és una poma
  - Una matriu d'enters és un enter
  - Un motor és un vehicle

- De totes maneres, pot haver casos en els quals aquest test pot fallar i no obstant això la relació d'herència és evident. Però, per a la major part dels casos, l'aplicació d'aquesta tècnica és adequada.

HERÈNCIA
Motivació: Principals usos

- ***L'herència com reutilització de codi:*** Una classe filla pot heretar comportament d'una classe pare, per tant, el codi no necessita tornar a ser escrit per a la filla. Açò fa que es reduïsca molt la quantitat de codi necessari per a desenvolupar una nova idea.
- ***L'herència com reutilització de conceptes:*** Açò ocorre quan una classe filla **sobrescriu** el comportament definit pel pare. Encara que no es comparteix codi entre la classe pare i la classe filla, les dues comparteixen la definició del mètode (comparteixen el concepte).

HERÈNCIA

- Ja hem comentat com la ment humana classifica els conceptes d'acord a dues dimensions:
  - Pertenència (TÉ-UN)
  - Varietat (ÉS-UN)
- L'herència aconsegueix classificar els tipus de dades (abstraccions) per varietat, acostant un poc més el món de la programació a la manera de raonar humana.
  - Aquesta manera de raonament humà es denomina GENERALITZACIÓ, i dóna lloc a jerarquies de generalització/especialització.
  - La implementació d'aquestes jerarquies en un llenguatge de programació dóna lloc a jerarquies d'herència. En elles tenim classes pare/superclasses i classes filla/subclasses.
- És una relació que normalment s'estableix entre CLASSES, no entre OBJECTES
  - Excepció: alguns llenguatges tardans han incorporat el concepte d'herència d'objectes (per raons de reutilització de codi). Així, un objecte individual pot ser estés amb capacitats d'altre objecte en temps d'execució.
    - Python, Ruby.

Herència com implementació de la Generalització

- La generalització és una relació semàntica entre classes, que determina que la interfície de la subclasse ha de incloure totes les propietats públiques i privades de la superclasse.
- L'herència és el mecanisme d'implementació mitjançant el qual elements més específics incorporen l'estructura i comportament d'elements més generals (Rumbaugh 99)

Herència com implementació de Generalització.

- Disminueix el nombre de relacions (associacions i agregacions) del model
- Augmenta la comprensibilitat, expressivitat i abstracció dels sistemes modelats.
- Tot açò a costa d'un major nombre de classes

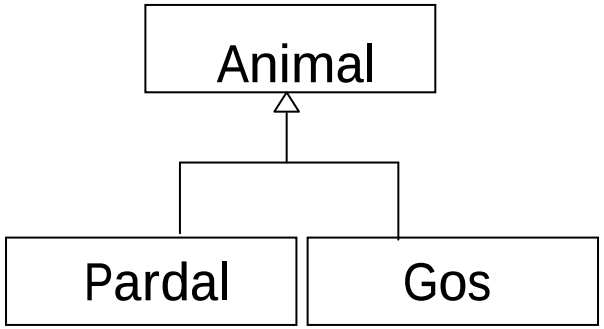
Tipus d'Herència

- Simple/Múltiple
- D'implementació/d'interfície

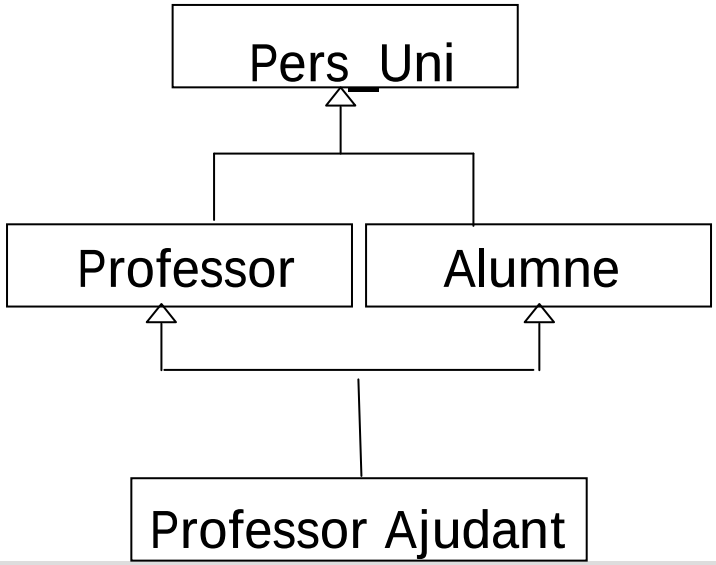
- A nivell semàntic, Bertrand Meyer distingueix 17 tipus d'herència.

Tipus d'Herència

- Simple/Múltiple
  - Simple: única classe base



  - Múltiple: Més d'una classe base



Tipus d'Herència

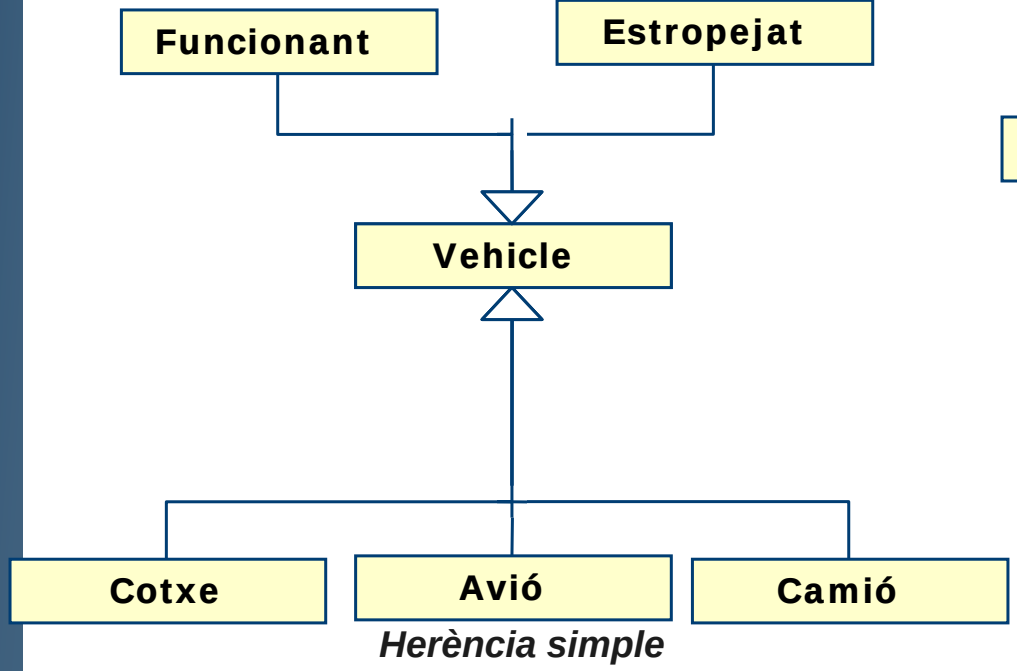
- D'implementació/d'interfície
  - D'implementació: La implementació dels mètodes és heretada. Pot sobreescriure-se en les classes filles.
  - D'interfície: Només s'hereta la interfície, no hi ha implementació a nivell de pare (interfície/implements en Java, classes abstractes en C++)

Herència
Caracterització

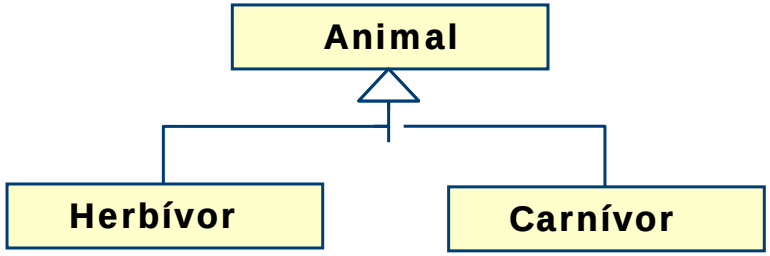
- Atributs de la generalització
  - Solapada/Disjunta
    - Determina si un objecte pot ser *al mateix temps* instància de dues o més subclasses d'eixe nivell d'herència.
    - C++ no soporta l'herència solapada (tipat fort)
  - Completa/Incompleta
    - Determina si totes les instàncies de la classe pare són *al mateix temps* instàncies d'alguna de les classes filles (completa) o, pel contrari, hi ha objectes de la classe pare que no pertanyen a cap subcategoria de les reflexades per les classes filles (incompleta).
    - A nivell d'implementació, una jerarqua d'herència completa sol implicar que la classe pare pot ser definida com abstracta (és a dir, impedir que es creen instàncias d'ella).
  - Estàtica/Dinàmica
    - Determina si un determinat objecte *pot pasar de ser instància d'una classe filla a altra* dins d'un mateix nivell de la jerarquia d'herència.
    - C++ no soporta l'herència dinàmica (tipat fort)

Herència
Caracterització

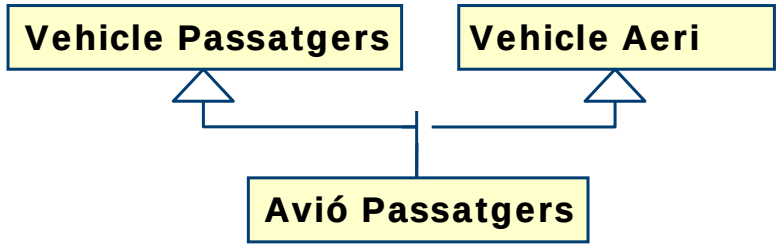
- Exemples



Herència simple

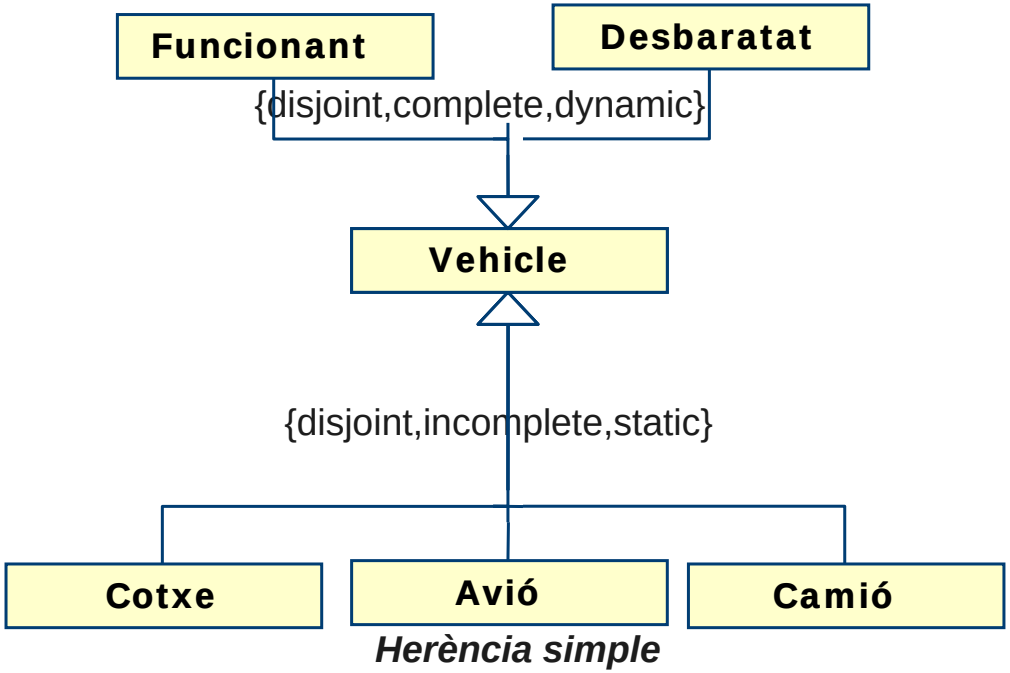


Herència simple

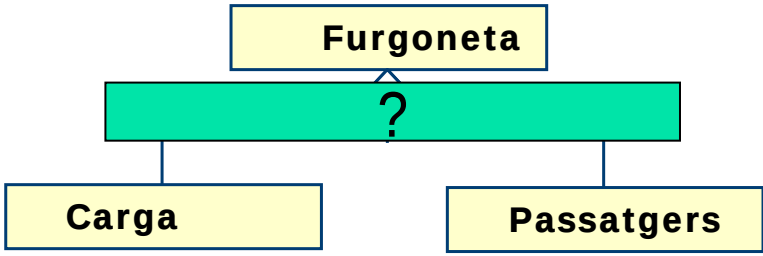


Herència múltiple

Herència
Caracterització: exemples



Herència simple



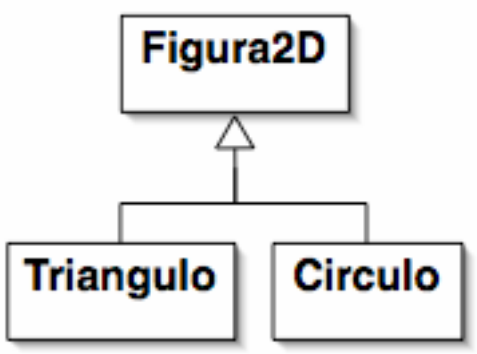
Herència simple

HERÈNCIA

Herència Simple

Herència Simple en C++

```
class Figura2D {
  public:
    Figura2D(Coordenada posicion, Color c);
    ...
    Color getColor();
    void setColor(Color c);
  private:
    Coordenada origen;
    Color colorRelleno;
};

class Circulo : Figura2D {
...
  public:
    void vaciarCirculo() {
      colorRelleno=CAP;
      // ¡ERROR! colorRelleno és privat
      setColor(CAP); // OK
    }
};
```




```
int main() {
  Circulo c;
  c.setcolor(AZUL);
  c.getColor();

  ...
}
```

Herència en C++

Visibilitat atributs/mètodes

- C++ introdueix un nou àmbit de visibilitat pel tractament de l'herència: **protected**
  - Les dades/funcions membres *protected* són privats per a totes aquelles classes no derivades i mètodes externs, però accesibles per a una classe derivada de la classe en la qual se ha definit la variable protegida.

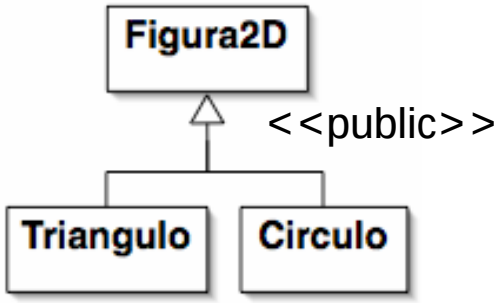
```
class Figura2D {
...
    protected:
      Color colorRelleno;
...
};

class Circulo : Figura2D {
   public:
     void vaciarCirculo() {
       colorRelleno=CAP; //OK, protected
     }
...
};
```

```
int main () {
  Circulo c;
  c.colorRelleno=NINGUNO;
 // ¡ERROR! colorRelleno
 // es privado aquí
}
```

Tipus d'Herència Simple (en C++)

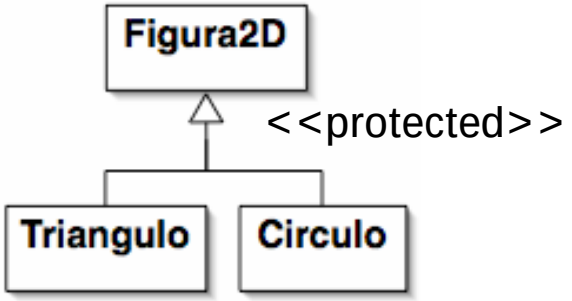
- Herència Pública (per defecte)

```
class Circulo : public Figura2D {
...
};
```



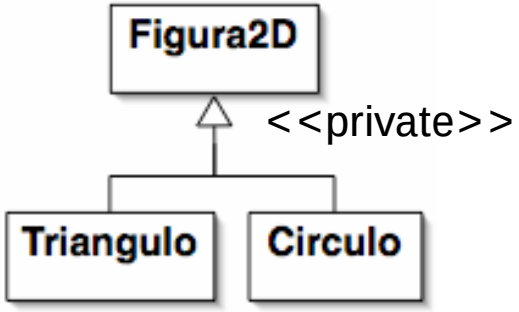
- Herència Protegida

```
class Circulo : protected Figura2D {
...
};
```



- Herència Privada

```
class Circulo : private Figura2D {
...
};
```

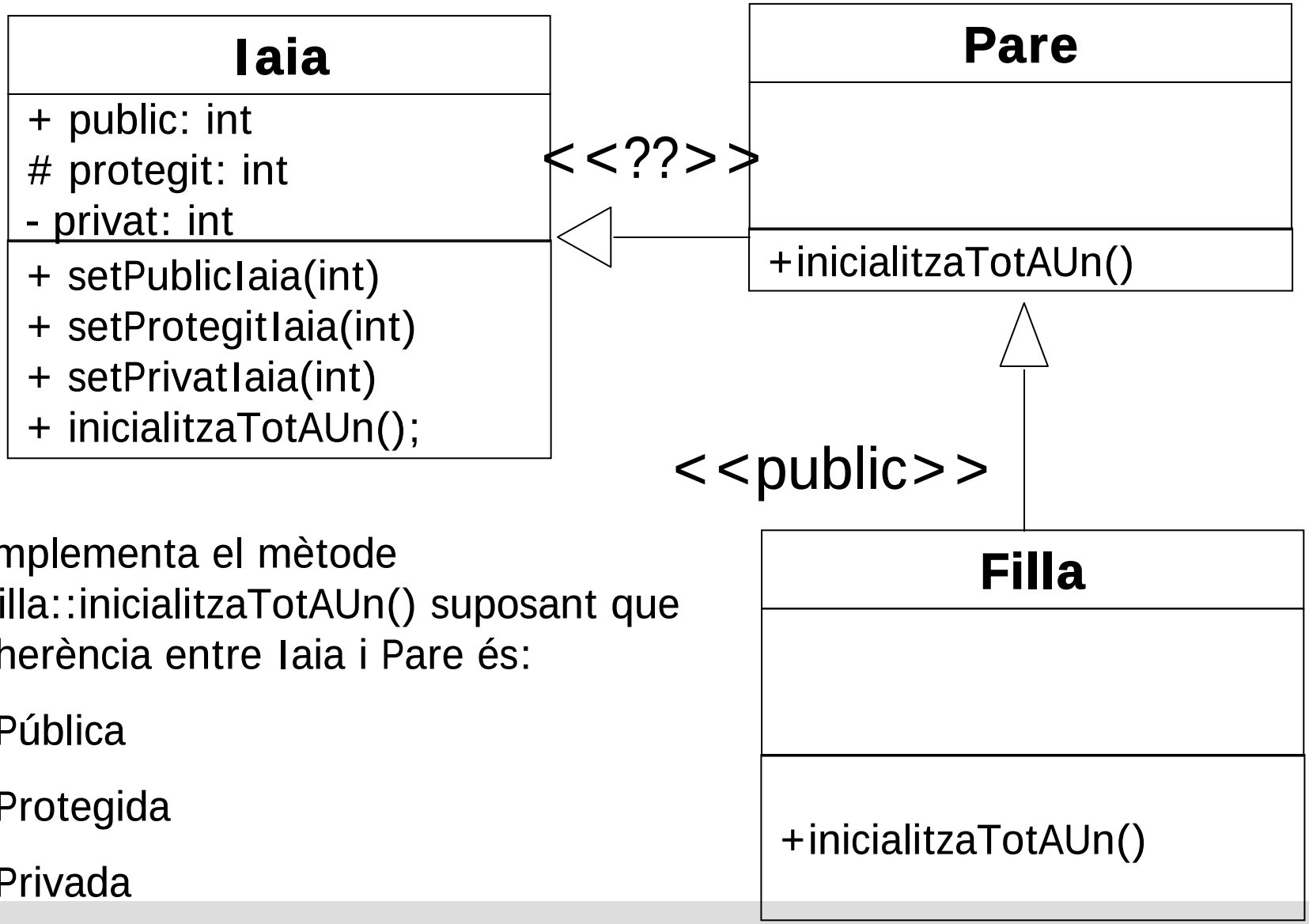


Tipus d'Herència Simple

Àmbit Herència / Visibilitat en classe base	CD (*) H. Pública	CD H. Protegida	CD H. privada
Private	No direct. accesible	No direct. accesible	No direct. accesible
Protected	Protected	Protected	Private
Public	Public	Protected	Private

(*) CD: Classe derivada

Tipus Herència Simple
Exercici



Implementa el mètode Filla::inicialitzaTotAUn() suposant que l'herència entre Iaia i Pare és:

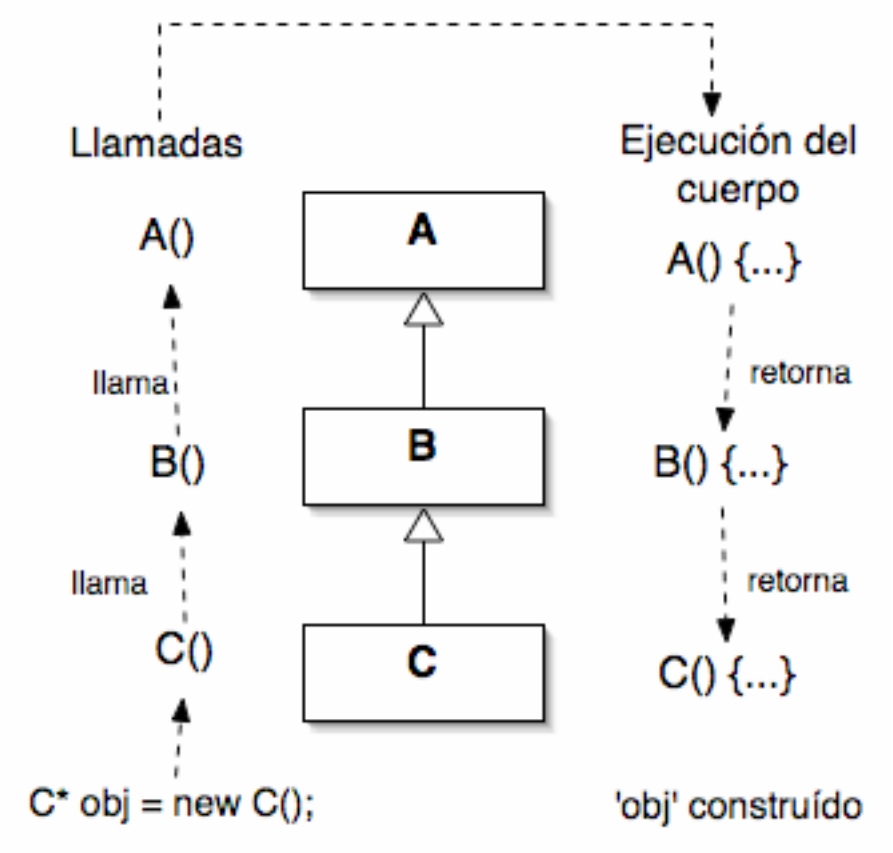
- Pública
- Protegida
- Privada

Herència Simple

- En la classe derivada es pot:
  - **AFEGIR** nous mètodes/atributs propis de la classe derivada
  - **Modificar** els mètodes heretats de la classe base
    - **REFINAR**: s'afegeix comportament nou abans i/o després del comportament heretat. (*Simula, Beta*) (es pot simular en C++, Java)
      - *C++, Java*: Constructors i destructors es refinen
    - **REEMPLAÇAR**: el mètode heretat es redefineix completament, de manera que substituïsca a l'original de la classe base.

El constructor en herència simple

- Els constructors no s' hereten
  - Sempre són definits per a les classes derivades
  - Creació d'un objecte de classe derivada: S'invoca a tots els constructors de la jerarquia
  - Ordre d'execució de constructors: Primer s'executa el constructor de la classe base i després el de la derivada.



El constructor en herència simple

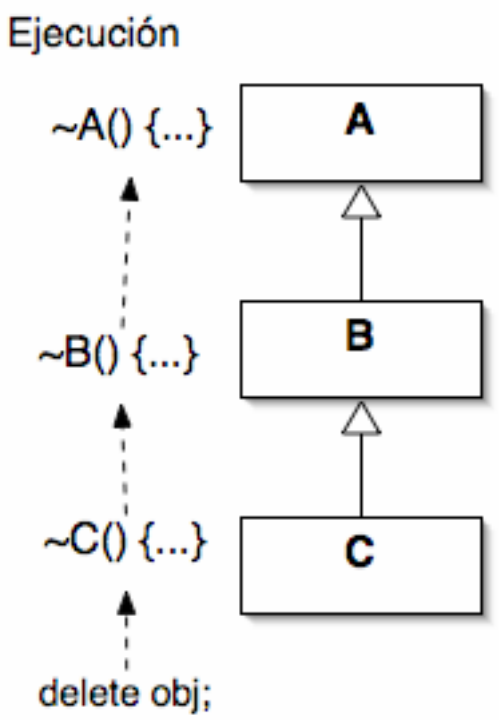
- Açò implica que la classe derivada aplica una política de **refinament**: afegir comportament al constructor del pare.

- Execució implícita del constructor per defecte de classe base al invocar a un constructor de classe derivada.
- Execució explícita de qualsevol altre tipus de constructor en la zona d'inicialització (refinament explícit). En particular, el constructor de còpia.

(CONSELL: Inicialització d'atributs de la classe base: en la classe base, no en la derivada)

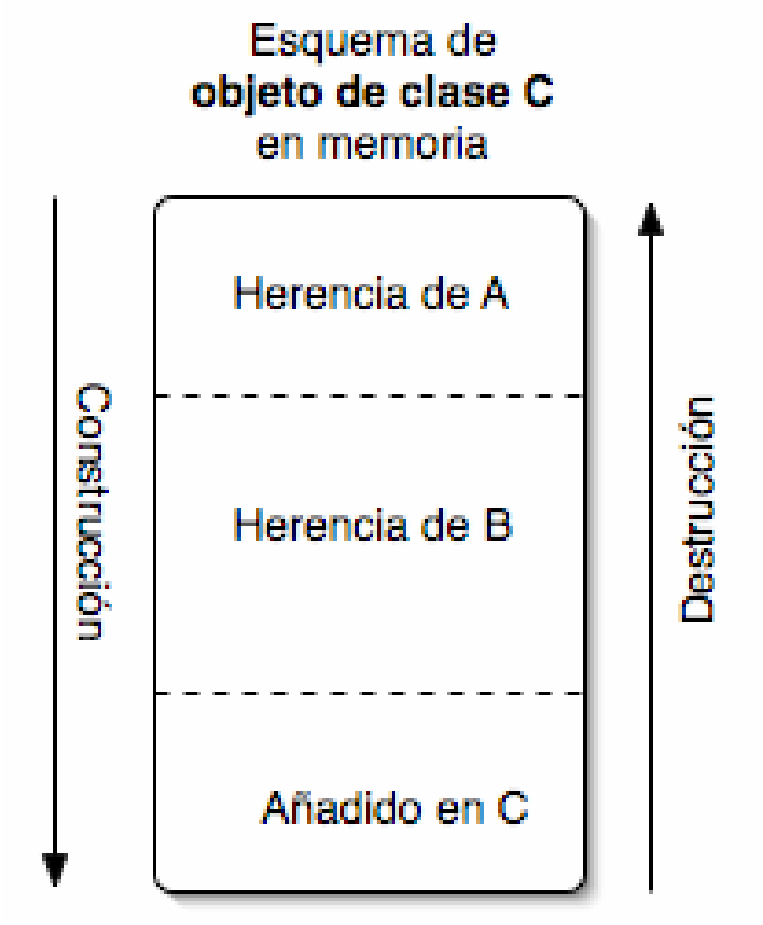
El destructor en herència simple

- El destructor no s'hereta.
  - Sempre és definit per a la classe derivada
  - Destrucció d'un objecte de classe derivada: s'invoca a tots els destructors de la jerarquia
  - Primer es destrueix l'objecte derivat i després l'objecte base.
  - Cridada implícita als destructors de la classe base.

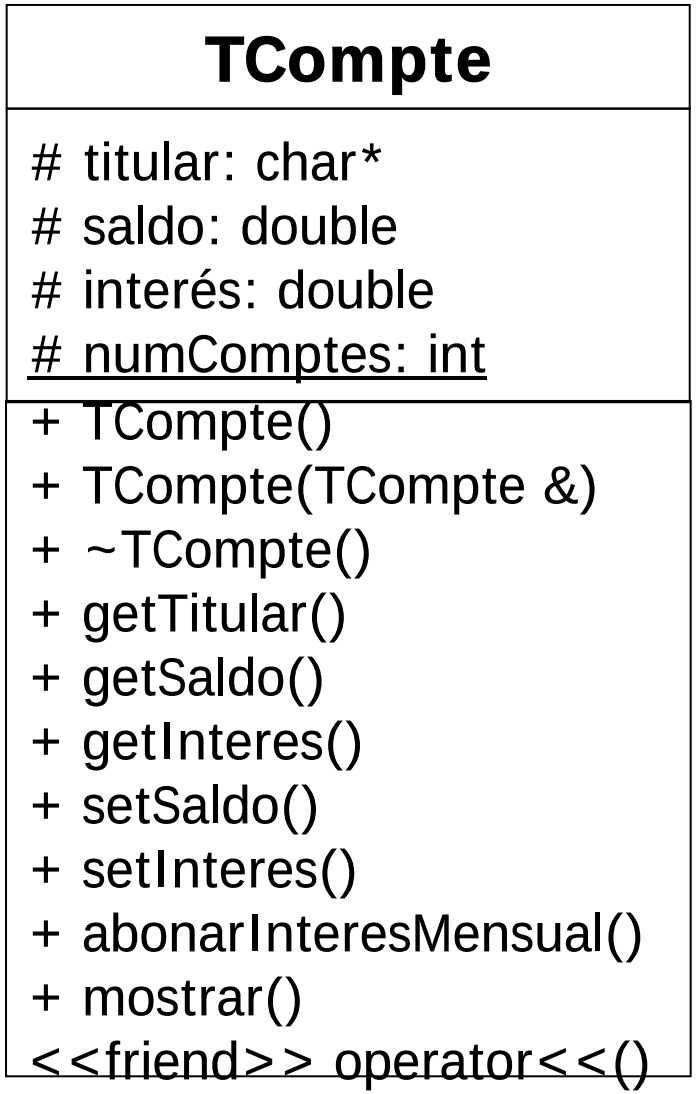


Construcció/destrucció en resum...

- Els objectes es destrueixen en ordre invers al de construcció.



Exemple Classe Base



HS (base): TCompte

```
class TCompte{
  protected:
   string titular;
   double saldo;
   double interes;
   static int numComptes;
   friend ostream& operator<<(ostream&, const TCuenta&);


  public:
  TCompte(string t, double s=0.0, double i=0.0)
      :titular(t), saldo(s), interes(i)
      { numComptes++; }
};
```

HS (base): TCompte (II)

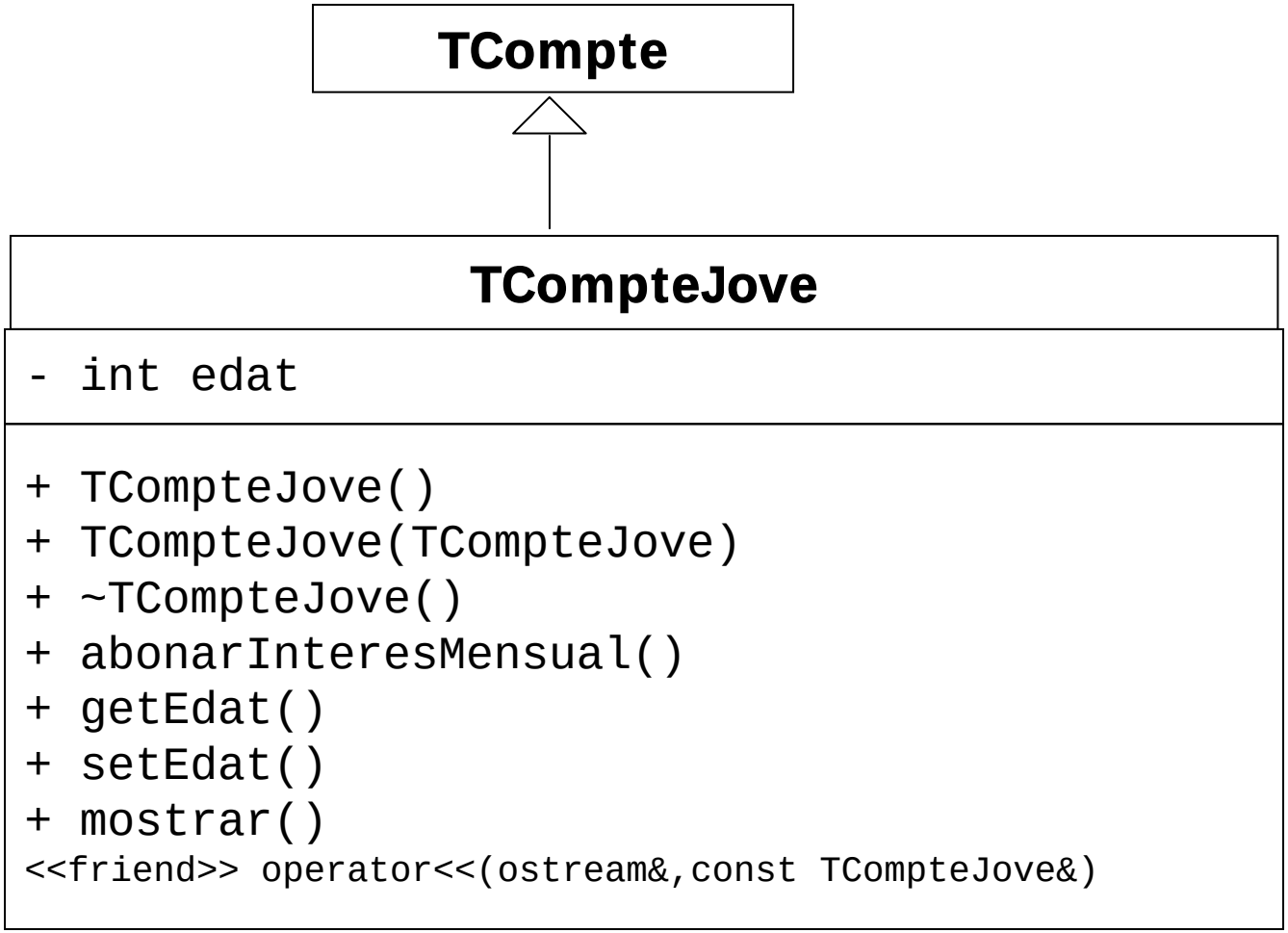
```
 TCompte(const TCompte& tc){
       titular=tc.titular;
       saldo=tc.saldo;
      interes=tc.interes;
       numComptes++;
}
~TCompte(){numComptes--;}

char* getTitular() const {return titular;};
double getSaldo() const {return saldo;};
double getInteres() const {return interes;};

void setSaldo(double s){saldo=s;};
void setInteres (double i){interes=i;};

void abonarInteresMensual(){
       setSaldo(getSaldo()*(1+getInteres()/100/12));};
```

HS (base): TCompte (III)

```
friend ostream& operator<< (ostream& os,
            TCompte& unCompte){
            os << “Titular="<<unCompte.titular<<endl;
            os << "Saldo="<<unCompte.saldo<<endl;
                        os << “Interes="<<unCompte.interes<<endl;
            os << “NumComptes="<<TCompte::numComptes<<endl;
            return os;}
void mostrar (){
            cout << “Titular="<<titular<<endl;
            cout << "Saldo="<<saldo<<endl;
            cout << “Interes="<<interes<<endl;
                        cout << “NumComptes="<<TCompte::numComptes;
                        cout<<endl;
             }
};//fi classe
```

- **Com puc simplificar la funció mostrar()?**

Exemple classe derivada



Herència Simple (derivada): TCompteJove (I)

```
class TCompteJove: public TCompte {
  private:
        int edat;
  public:
   TCompteJove(string unNomb,int unaEdat,
    double unSaldo=0.0, double unInteres=0.0)
    : TCompte(unNomb,unSaldo,unInteres), edat(unaEdat)
    { edat=unaEdat; }

  TCompteJove(const TCompteJove& tcj)
  // cridada explícita a constructor de còpia de TCompte.
      : TCompte(tcj), edat(tcj.edat) { }

  ~TCompte() { edat=0; }

  TCompte& operator=(const TCompteJove& tcj) {
    if (this!=&tcj) {
      TCompte::operator=(tcj);
      edat = tcj.edat;
    }
    return *this;
   }
```

S'ha d'incrementar numComptes?

Refinament

Herència Simple (derivada): TCompteJove (II)

```
void abonarInteresMensual() {
      //no interés si el saldo es inferior al límit
      if (getSaldo()>=10000) {
        setSaldo(getSaldo()*(1+getInteres()/12/100));
    }
}
```

Reemplaçament

HS (derivada): TCompteJove (II)

```
  int getEdat(){return edat;};
  void setEdat(int unaEdat){edat=unaEdat;};

  void mostrar(){
        TCompte::mostrar();
        cout<<“Edat:”<<edat<<endl;
  }
  friend ostream& operator<< (ostream& os, TCompteJove& c){
        os << “Titular="<<c.titular<<endl;
        os << "Saldo="<<c.saldo<<endl;
        os << "Edat="<<c.edat<<endl;
        os << “NumComptes="<<TCompteJove::numComptes<<endl;
        return os;
  }
};//fin classe TCompteJove
```

Mètodes afegits

Mètode Refinat

- **Com podríeu definir l’operador << per a que siga inmune als canvis en la classe base i a més ocupara menys línies de codi**?
- **Què està ocorrent amb el nombre de Comptes? És necessari incrementar el numComptes?**

En la classe TCompteJove...

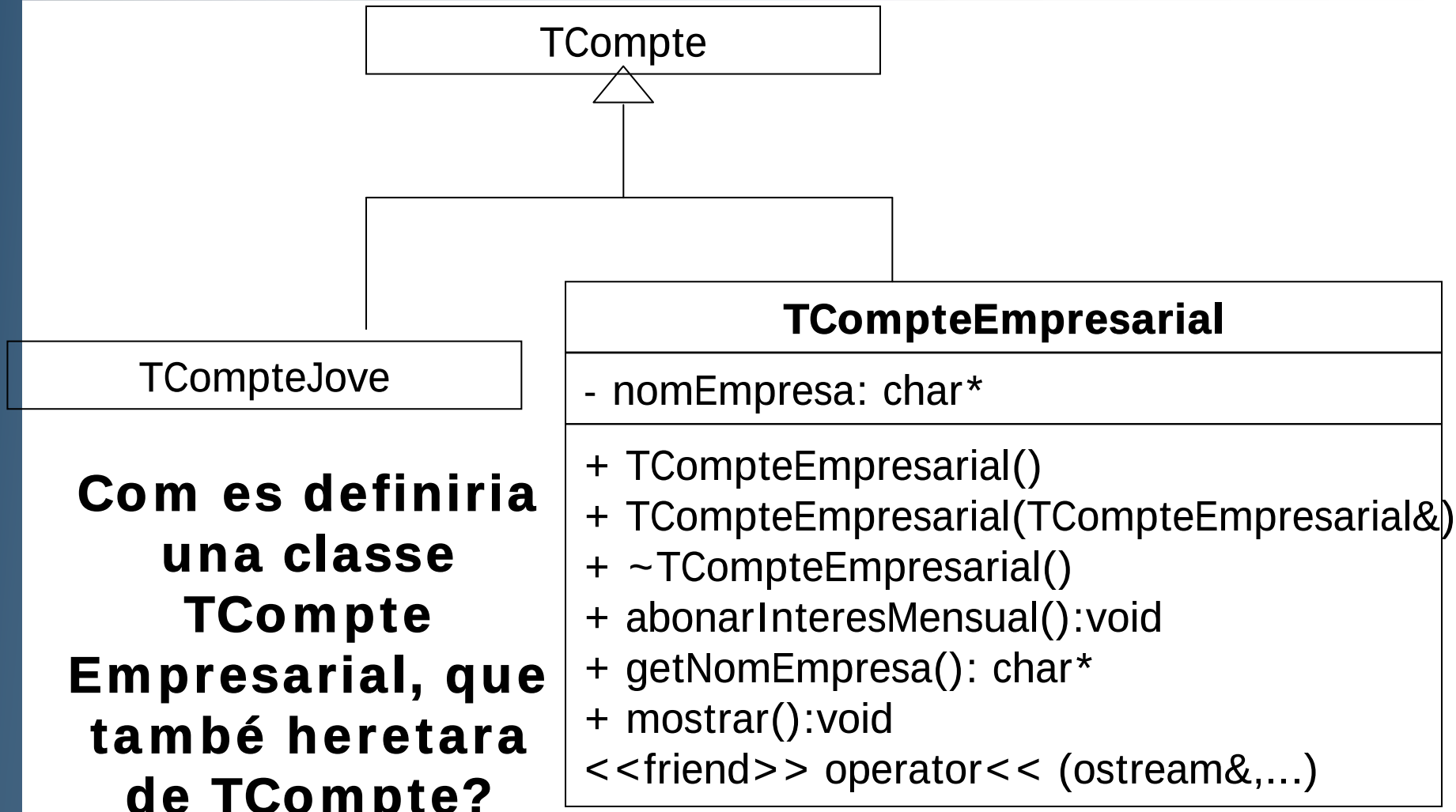
Si intente cridar directament al destructor de la classe base...

```
~TCompteJove(){
  cout<<“Cride dest de TCompteJove"<<endl;
  TCompte::~TCompte();
}
```

Error de compilació: cannot call destructor TCompte::~TCompte() without object.

- This (de tipo TCompteJove) no serveix com argument del mètode TCompte, el què implica que **el ppi de sustitució no s’acompleix amb el destructor.**

Exercici: TCompteEmpresarial



Com es definiria una classe TCompte Empresarial, que també heretara de TCompte?

Particularitats Herència

- Com ja hem vist, en les jerarquies d'herència hi ha un refinament implícit de:
  - Constructors (normal, de còpia)
  - Destructors
- Tb. Es dona una compartició de:
  - Variables estàtiques
- L'operador d'assignació (Tema 4) tampoc es manté com mètode fundacional (i.e. reutilitzable sense canvis) en la jerarquia, sinó que se substitueix sempre per un propi de la classe derivada (pot ser d'ofici, proporcionat pel compilador).
- Per aquest motiu la forma canònica de la classe implica sempre definir aquestes quatre funcions membre

- **S'hereten les funcions amigues?**

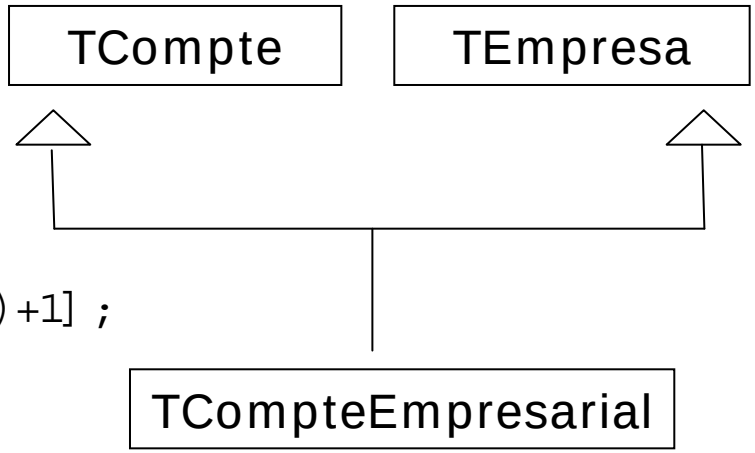
Exercici Propost

- Definiu una jerarquia d'herència en la que es contemplen els empleats d'una empresa, emmagatzemant el seu nom i el seu sou, i els gerents com un tipus especial d'empleats dels quals emmagatzemarem el departament i el nom de la seua secretària. Realitzar l'especificació UML i el .h de les classes implicades.

HERÈNCIA

Herència Múltiple

Exemple Herència Múltiple



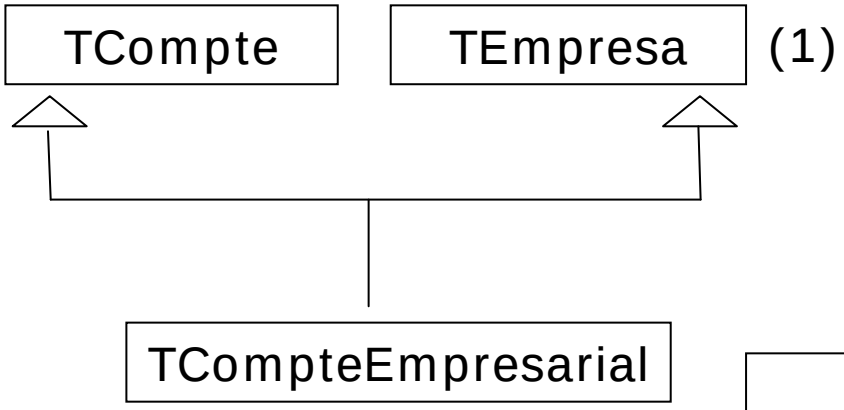

```
class TEmpresa{
 protected:
    char *nomEmpresa;
 public:
   TEmpresa(const char* unaEmpresa) {
    nomEmpresa=new char[strlen(unaEmpresa)+1] ;
    strcpy(nomEmpresa,unaEmpresa);
   };

   void setNombreEmpresa(char *nouNomb) {
    if (nomEmpresa!=NULL)
      delete [] nomEmpresa;
    nomEmpresa=new char[strlen(nuevoNombre)+1];
    strcpy(nomEmpresa,nouNomb);}

   ~TEmpresa() { delete [] nomEmpresa; }
};
```

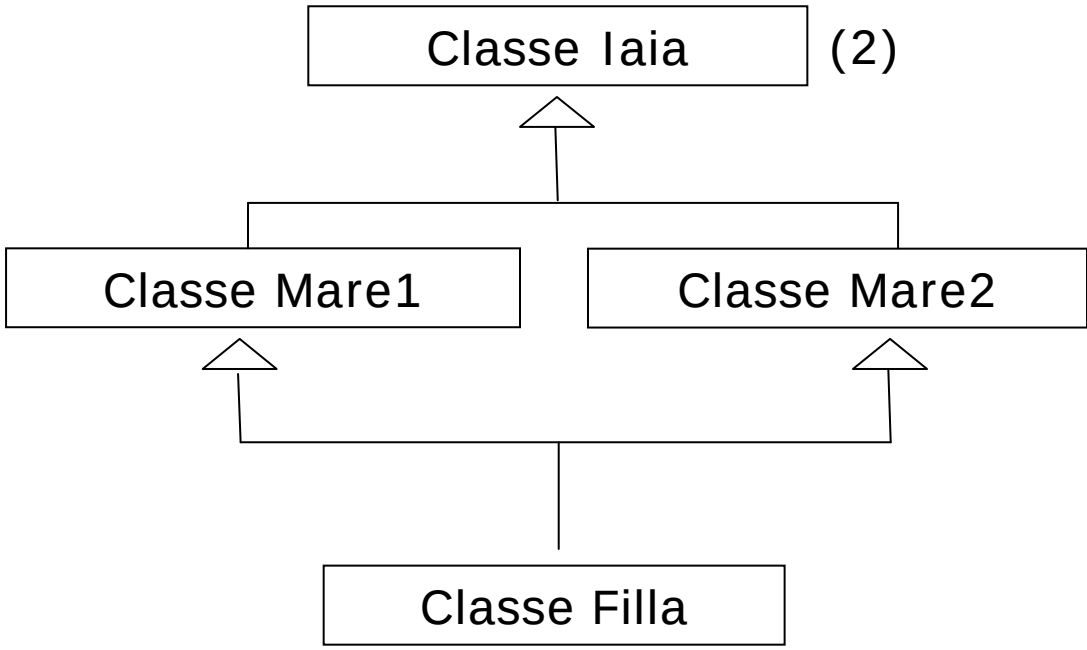
Com implementar TCompteEmpresarial?

Exemple Herència Múltiple (II)

```
class TCompteEmpresarial
: public TCompte, public TEmpresa {

    public:

     TCompteEmpresarial(const char *unNombreCompte,
            const char *unNombEmpresa,
            double unSaldo=0, double unInteres=0)
       :TCompte(unNombCompte,unSaldo,unInteres),
            TEmpresa(unNombEmpresa)
     {};
};
```

Problemes en Herència Múltiple





Què problemes poden donarse en (1)? I en (2)?

Ús de Àmbit en Herència Múltiple

```
class TCompteEmpresarial: public TCompte, public
    TEmpresa{
   string n;
    ...
    if ...
     n= TCompte::getNomb();
    else
      n= TEmpresa::getNomb();
}
```

Herència virtual (Herència múltiple)

```
class Mare_1: virtual public Iaia{
...
}

class Mare_2: virtual public Iaia{
...
}

Class Filla: public Mare_1, public Mare_2 {
    ...
    Filla() : Mare_1(), Mare_2(), Iaia(){
    };
}
```

HERÈNCIA

Herència d’Interfície

Herència d'interfície

- L'herència d'interfície NO hereta codi
- És una herència sense efectes secundaris.
- S'utilitza exclusivament amb el propòsit de garantir la **sustituibilitat.**

Ppi de sustitució de Liskow

- El Principi de Sustitució de Liskow (1987) afirma que:

  "Ha de ser possible utilitzar qualsevol objecte instància d'una subclasse en el lloc de qualsevol objecte instància de la seua superclasse sense que la semàntica del programa escrit en els termes de la superclasse es veja afectat."

- Les classes que compleixen este principi es diu que són SUBTIPUS a més de ser subclasses.
- Tret especialment rellevant en llenguatges fortament tipats.
  - Estos llenguatges permeten a nivell sintàctic aplicar sempre este principi. És labor del dissenyador assegurar-se que efectivament té sentit en les jerarquies que defineix.

El principi de sustitució

- Tots els LOO soporten subtipus.
  - Llenguatges fortament tipats (tipat estàtic)
    - Caracteritzen els objectes per la seua classe
  - Llenguatges debilment tipats (tipat dinàmic)
    - Caracteritzen els objectes pel seu comportament

Llenguatge fortament tipat:

```
funcion medir(objeto: Medible)
{...}
```

Llenguatge debilment tipat:

```
funcion medir(objeto) {
si (objeto <= 5)
sino si (objeto == 0)
...}
```

El principi de sustitució

- **C++:** subtipus només a través de punters o referències

```
class Dependiente {
public:
int cobrar();
void darRecibo();
...};
class Panadero
: public Dependiente
{...}
Panadero p;
Dependiente& d1=p; // sustit.
Dependiente* d2=&p; // sustit.
Dependiente d3=p; // NO sustit.
```

- **Java:** directament

```
class Dependiente {
public int cobrar();
public void darRecibo();
...};
class Panadero
extends Dependiente
{...}
Panadero p = new Panadero();
Dependiente d1=p; // sustit.
```

Herència d'interfície

- **Objectius:**
  - Reutilització de conceptes (interfície)
  - Garantir que s'acompleix el principi de sustitució
- Implementació mitjançant classes abstractes (C++) o interfícies (Java/C#)

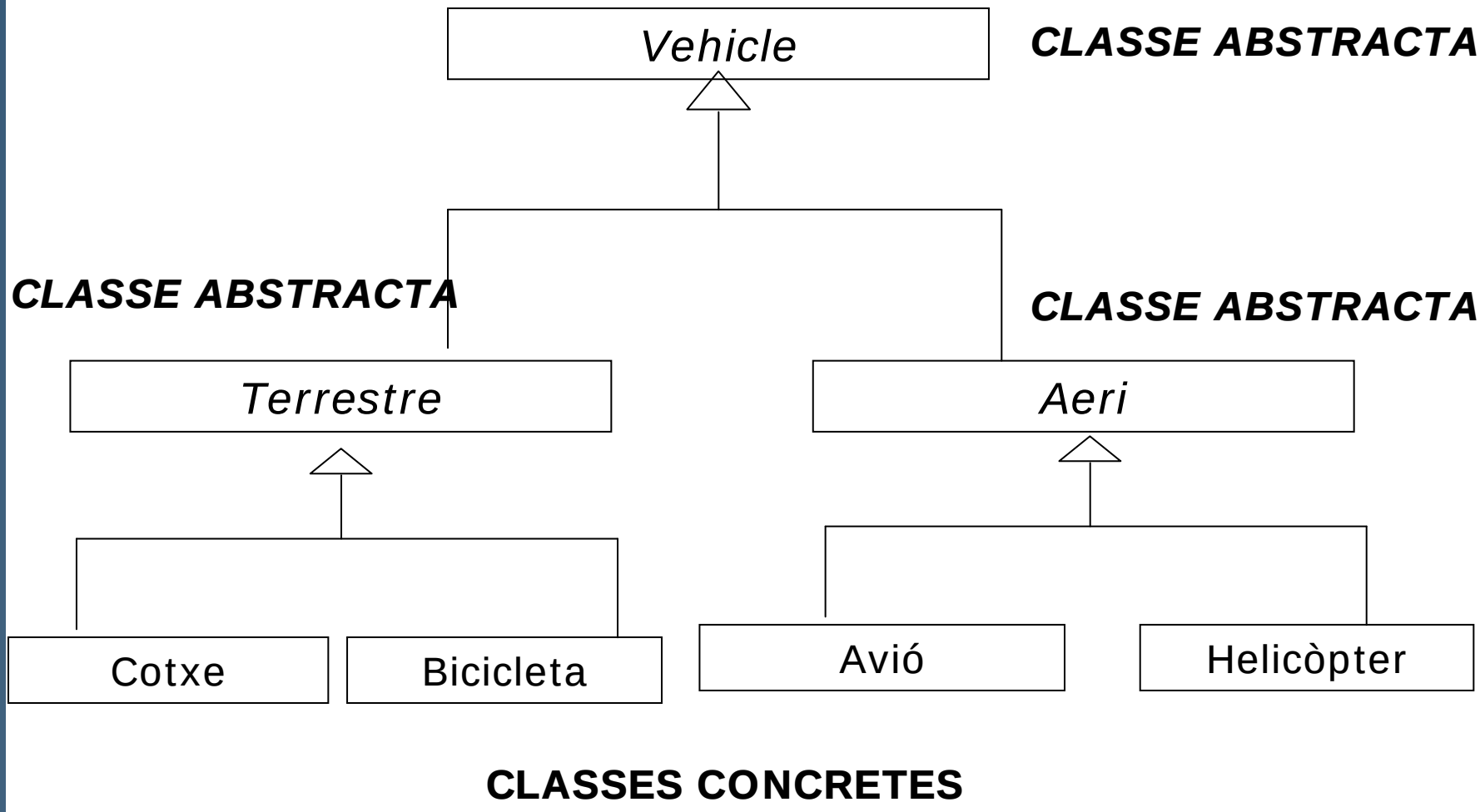
Herència d'interfície

- **Classes abstractes**
  - Són classes en les quals algun dels seus mètodes no està definit (mètodes abstractes)
  - No es poden crear objectes d'estes classes.
  - Sí es poden crear referències (o punters) a objecte d'una classe abstracta (que apuntaran a objectes de classes derivades)
    - Propòsit: Garantir que les classes derivades proporcionen una implementació pròpia de certs mètodes.
    - Sustituibilitat: Es garanteix la sustituibilitat.

Herència d'interfície

- Classes abstractes
  - Les classes que deriven de classes abstractes (o interfícies) han d'implementar tots els mètodes abstractes (o seran al mateix temps abstractes)
  - La classe derivada implementa la interfície de la classe abstracta

Classes Abstractes
Exemple



Herència d'interfície

- Classes abstractes en C++
  - Classes que contenen al menys un **mètode virtual pur** (mètode abstracte)

virtual <tipus retorn> metode(<llista args>) **= 0;**

Classe abstracta

```
class Forma
{
int posx, posy;
public:
virtual void dibujar()= 0;
int getPosicionX()
{ return posx; }
...
}
```

Classe derivada

```
class Circulo : public
Forma
{
int radio;
public:
void dibujar() {...};
...
}
```

Herència d'interfície

- Classes abstractes en Java

```
abstract class {
…
abstract <tipus retorn> metode (<llista args>);
}
```

Classe abstracta

```
abstract class Forma
{
private int posx, posy;
public abstract void dibujar();
public int getPosicionX()
{ return posx; }
...
}
```

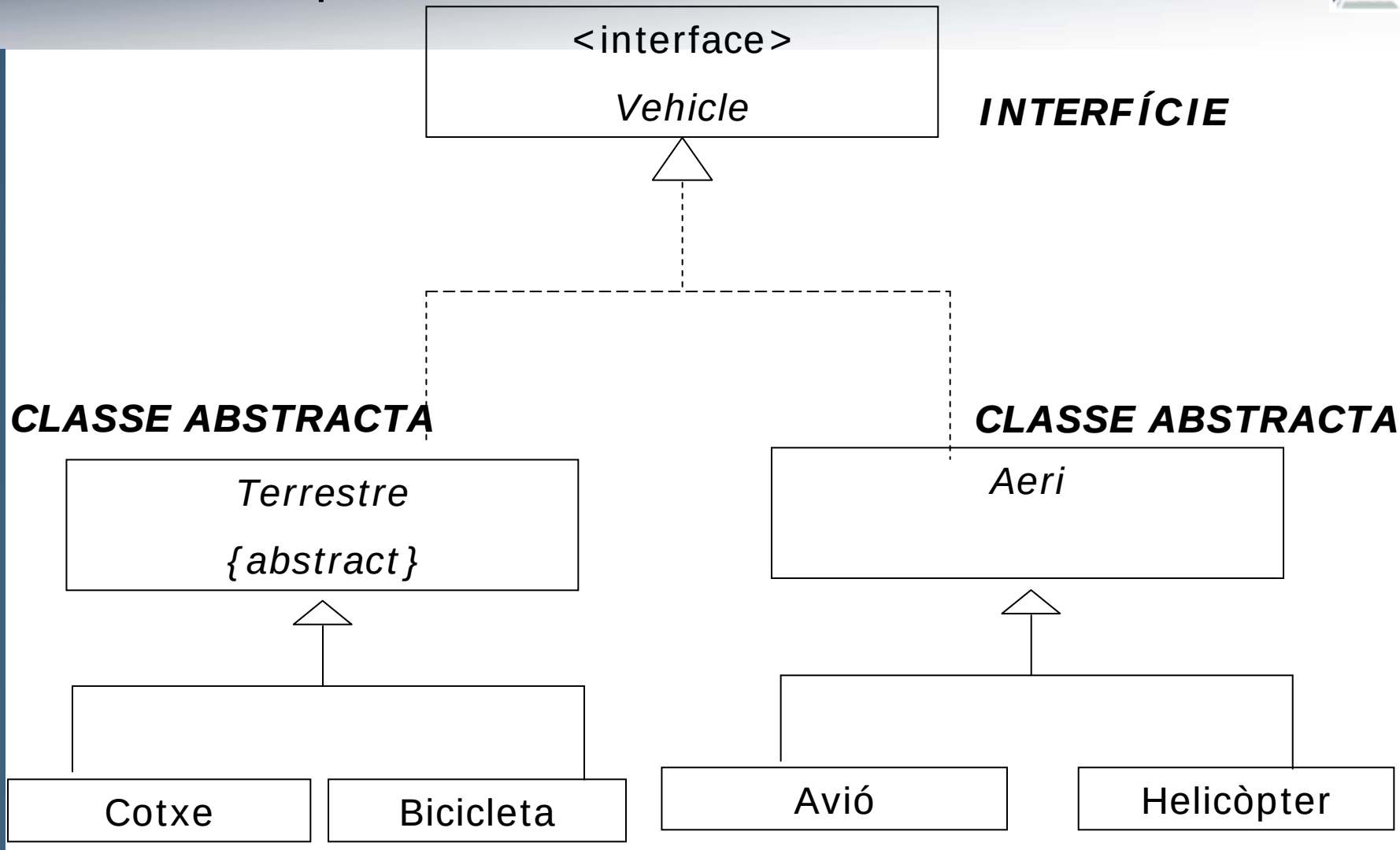
Classe derivada

```
class Circulo extends Forma {
private int radio;
public void dibujar()
{...};
...
}
```

Herència d'interfície

- **Interfícies**
  - Declaració d'un conjunt de mètodes sense definir.
  - En C++, són classes abstractes on cap mètode està definit.
  - Java/C#: declaració explícita d'interfícies
    - Las classes poden implementar més d'una interfície (herència múltiple d'interfícies)

Notació UML per a interfícies



Herència d'interfície

- Interfícies en Java

```
interface Forma
{
// - Sense atributs d'instància
// - Només constants estàtiques
// - Tots els mètodes són abstractes
void dibujar();
int getPosicionX();
...
}
```

```
class Circulo implements Forma
{
private int posx, posy;
private int radio;
public void dibujar()
{...};
public int getPosicionX()
{...};
}
```

Herència d'interfície

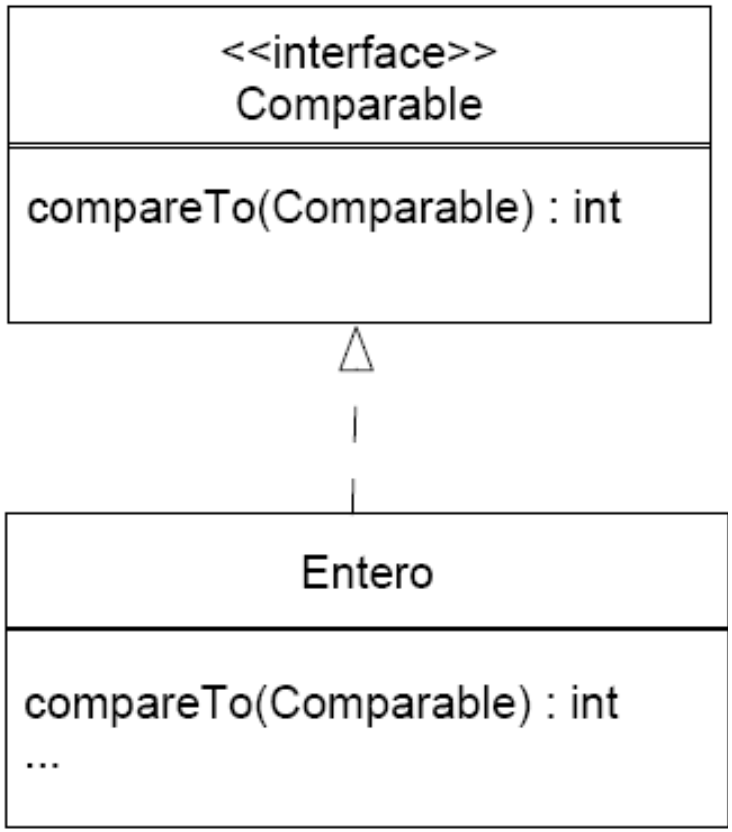
- Interfícies en C++

```
class Forma
{
// - Sense atributs d'instància
// - Només constants estàtiques
// - Tots els mètodes són abstractes
void dibujar()=0;
int getPosicionX()=0;
// la resta de mètodes virtuals purs...
}
```

```
class Circulo : public Forma // Herència pública
{
private:
int posx, posy;
int radio;
public:
void dibujar() {...}
int getPosicionX() {...};
}
```

Herència d'interfície

- Exemple d'interfície (Java)

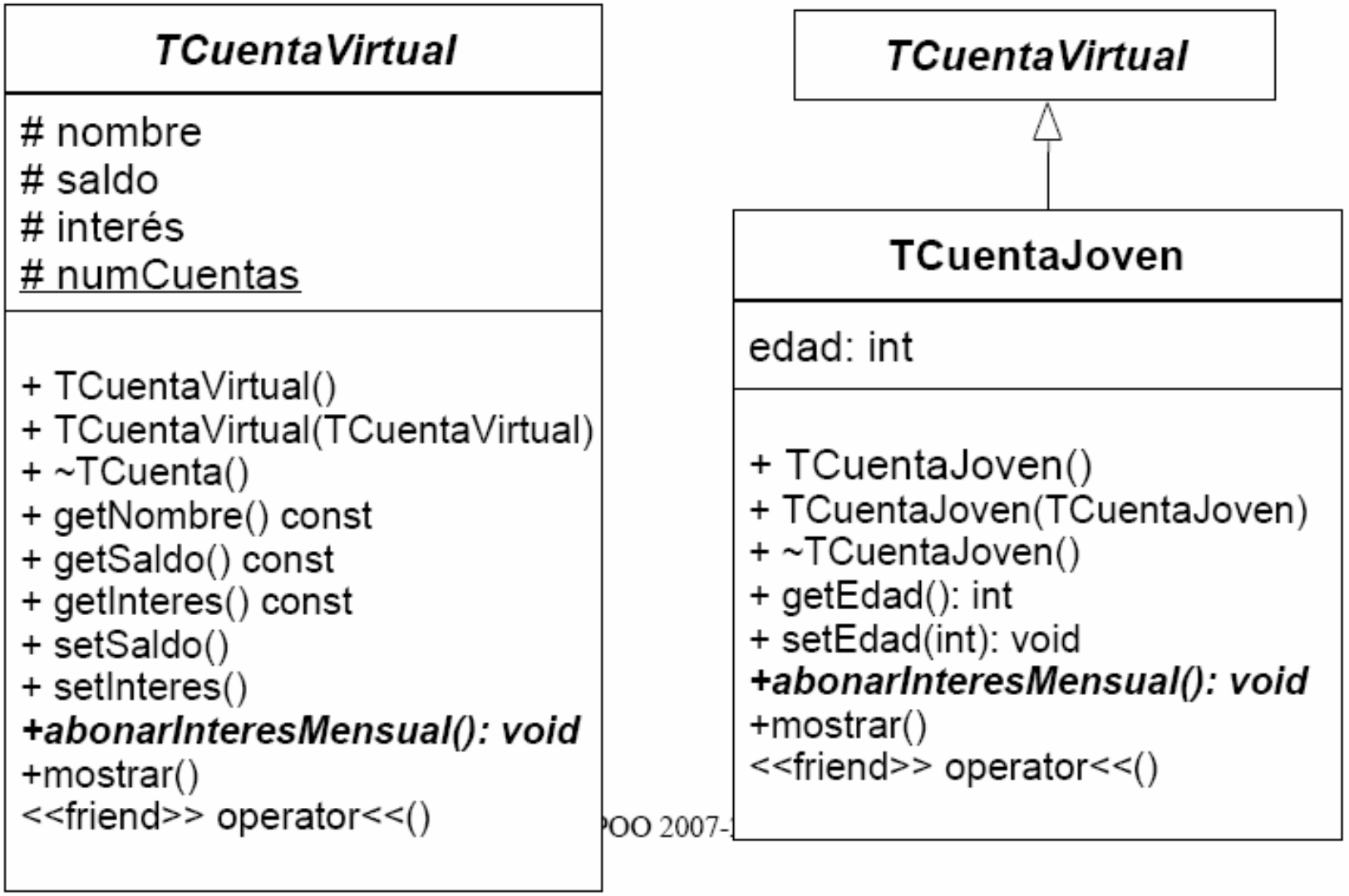



```
interface Comparable {
  int compareTo(Comparable o);
}
```

```
class Entero implements Comparable {
  private int n;

  Entero(int i) { n=i; }

  public int compareTo(Comparable e) {
      Entero e2=(Entero)e;
      if (e2.n > n) return -1;
      else if (e2.n == n) return 0;
      return 1;
  }
}
```

Herència d'interfície

- Exemple de classe abstracta (C++)



Herència d'interfície

```
class TCuentaVirtual {
  protected:
     ...
  public:
     ...
     virtual void abonarInteresMensual()=0;
};
```

```
class TCuentaJoven: public TCuentaVirtual{
    private:
        int edad;
    public:
      ...
      // IMPLEMENTACION
      void abonarInteresMensual(){
        //no interés si el saldo es inferior al límite
        if (getSaldo()>=10000){
           setSaldo(getSaldo()*(1+getInteres()/12/100));
        }
      };
// sigue...
```

Heréncia d'interfície

```
// sigue...
     void mostrar(){
       TCuentaVirtual::mostrar();
       cout<<"Edad:"<<edad<<endl;
     }

     friend ostream& operator<<(ostream& os, TCuentaJoven& c){
       // os<<(TCuentaVirtual)c<<endl;
       // ERROR: cast a clase abstracta

       c.TCuentaVirtual::mostrar();
       // Problema: mostrar() usa cout

       os << "Edad="<<unaCuenta.edad<<endl;
       return os;
     }
```

- (Troba la forma de reutilitzar el codi de l'operador d'eixida de la classe TCuentaVirtual en l'operador d'eixida de TCuentaJoven)

Mètodes virtuals i sobreescriptura

```
...
TCuentaVirtual* tcv;
tcv = new TCuentaJoven;
tcv->abonarInteresMensual();
// Llamada a TCuentaJoven::abonarInteresMensual()
delete tcv; // ¿?
...
```

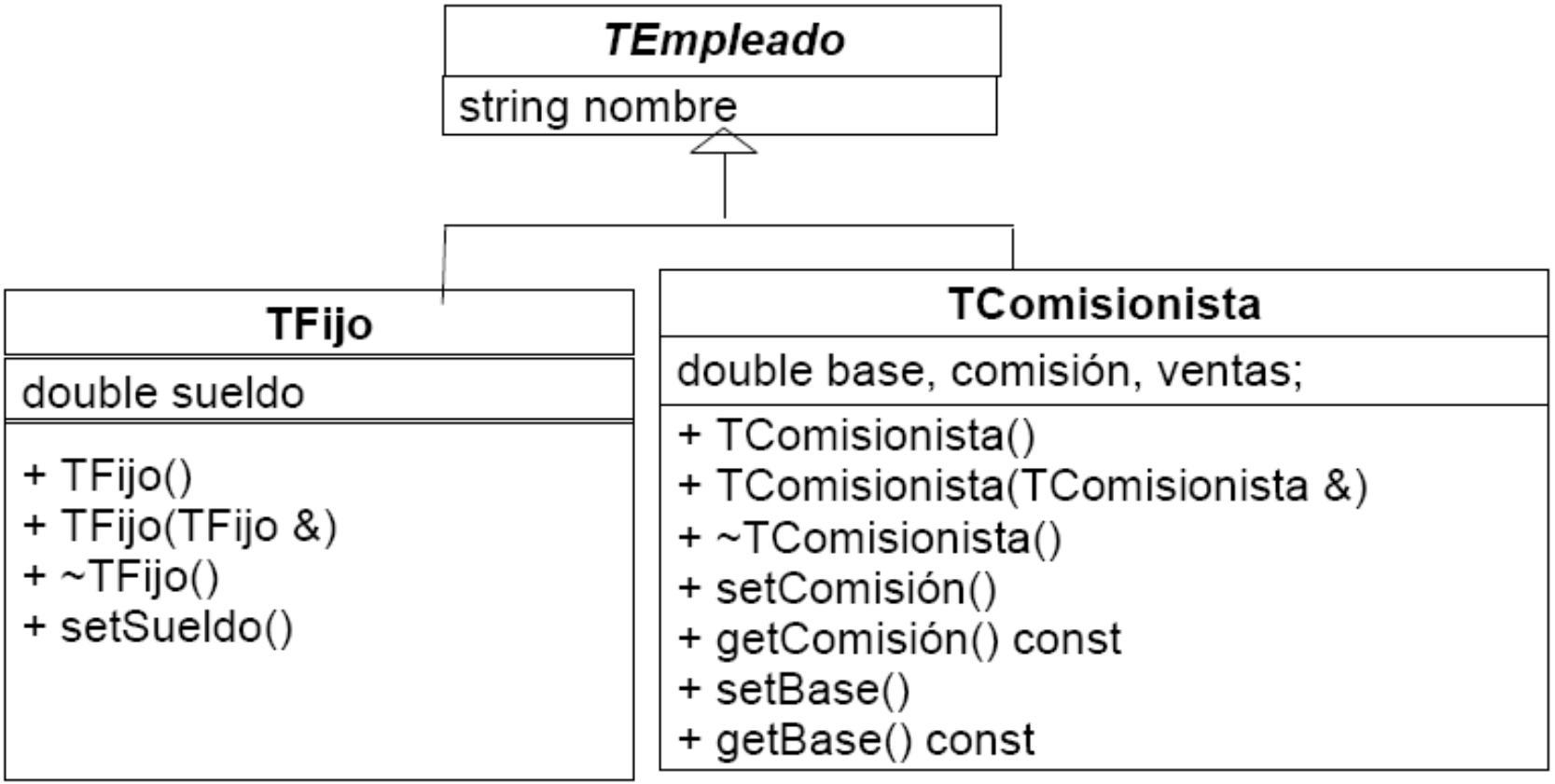
- La classe derivada redefineix el comportament de la classe base
- Es pretén invocar a certs mètodes redefinits des de referències a objectes de la classe base (principi de sustitució).
- En C++ és necessari indicar explícitament que eixa sustitució és posible mitjançant la definició d'eixos mètodes com virtuals (mitjançant la paraula clau **virtual**).
- En molts llenguatges OO este tret és suportat de forma natural:
  - Per exemple, en Java, els mètodes són virtuals per defecte
- Sobreescriptura: redefinició de mètodes virtuals en classes derivades

Mètodes virtuals i sobreescriptura

```
class TCuentaVirtual {
  virtual void abonarInteresMensual();
  virtual void setInteres(float i);
  virtual ~TCuentaVirtual();
}
```

- Si una classe té mètodes virtuals
  - És possible utilitzar el principi de sustitució:
  - Heretar d'ella i sobreescriure els mètodes virtuals
  - El destructor d'esta classe base ha de declarar-se com mètode virtual.

Exercisi: Pagament de nòmines



Exercisi: Pagament de nòmines

- Implementa les classes anteriors afegint un mètode getSalario(), que en cas de l'empleat fixe retorne el sueldo i en el cas del comisionista retorne la base més la comisió, de manera que el següent codi permeta obtindre el salari d'un empleat independentment del seu tipus.

```
int main(){
	int tipo;
	Empleado *eptr;
	cout<<"Introduce tipo"<<endl;
	cin>>tipo; //1:fijo, 2 comisionista
	switch (tipo){
		case 1: eptr=new Fijo();
		break;
		case 2:eptr=new Comisionista();
		break;
	}
	eptr->getSalario();
	delete eptr;
}
```

HERÈNCIA

Herència d'Implementació

Herència d’implementació

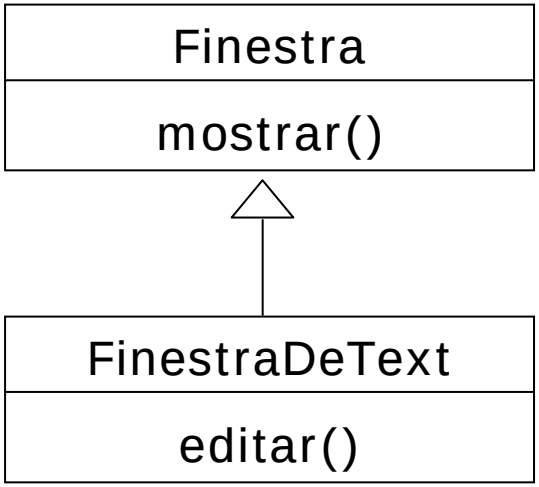
- Habilitat perquè una classe herete part o tota la seua implementació d'altra classe.
- Ha de ser utilitzada amb compte

Ús segur de l’herència d’implementació

- En l'herència existeix una tensió entre expansió (addició de mètodes més específics) i contracció (especialització o restricció de la classe pare)
- Aquesta tensió està en la base del seu poder, i també dels problemes associats amb el seu ús.
- En general, la redefinició de mètodes només hauria d'usar-se per a fer les propietats més específiques
  - Constrenyir restriccions
  - Estendre funcionalitat

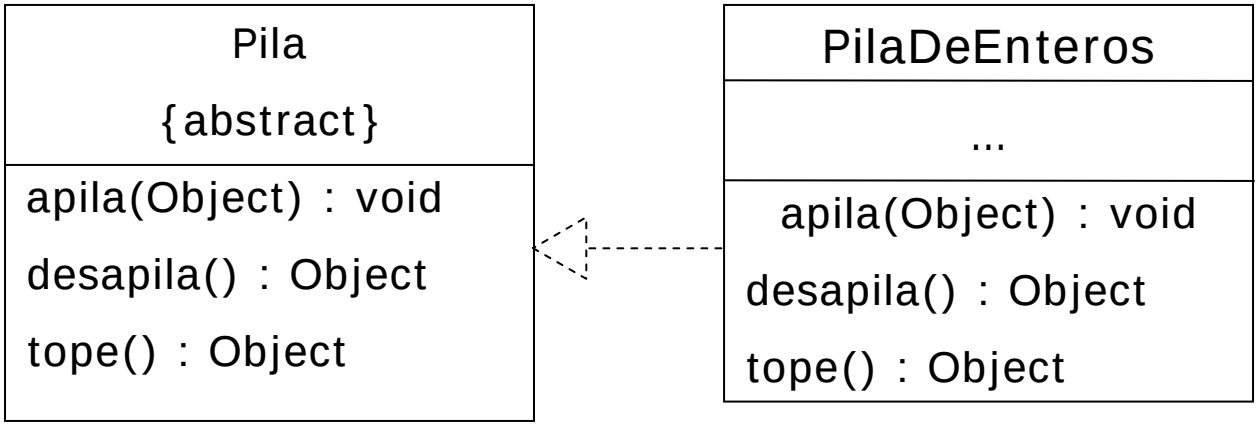
Ús segur de l'herència d'implementació

- **Especialització**
  - La classe derivada és una **especialització** de la classe base: afegeix comportament però no modifica res
    - Satisfà les especificacions de la classe base
    - Es compleix el principi de substitució (subtipus)



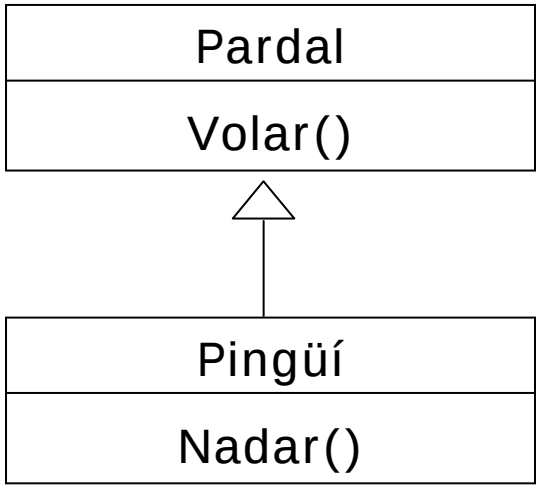
Ús segur de l'herència d'implementació

- **Especificació**
  - La classe derivada és una **especificació** d'una classe base abstracta o interfície.
    - Implementa mètodes no definits en la classe base (mètodes abstractes o diferits).
    - No afegeix ni elimina res.
    - La classe derivada és una realització (o implementació) de la classe base.



Ús insegur de l'herència d'implementació

- **Restricció (limitació)**



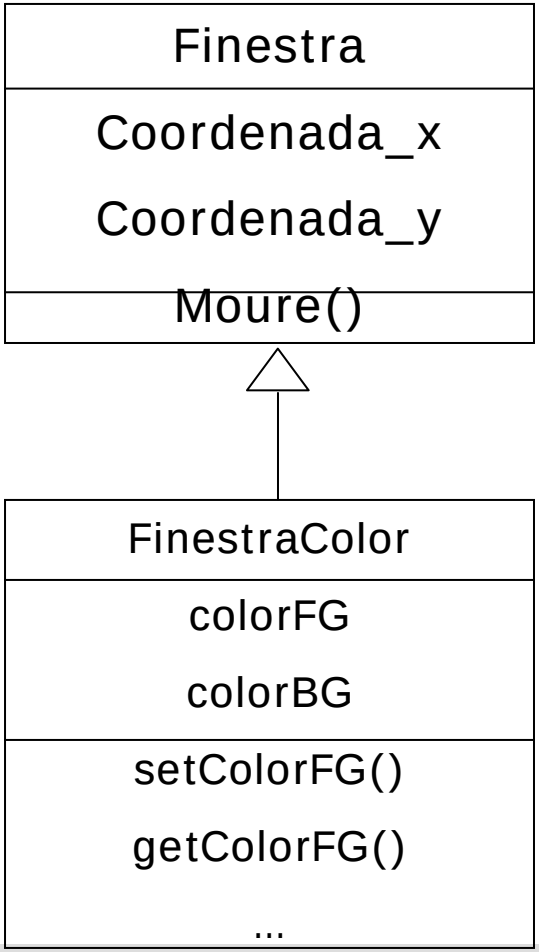
No totes les coses de la classe base serveixen a la derivada.

S'han de redefinir certs mètodes per a eliminar comportament present en la classe base

No es compleix el principi de substitució (un pingüí no pot volar)

Ús insegur de l'herència d'implementació

- **Generalització**

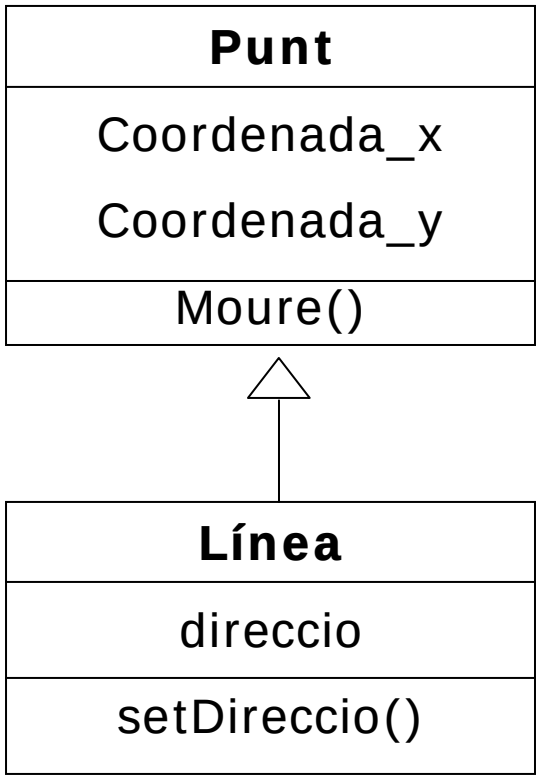


• S'estén el comportament de la classe base per a obtindre un tipus d'objecte més general.

• Usual quan no es pot modificar la classe base. Millor invertir la jerarquia.

Ús insegur de l'herència d'implementació

- **Variància (herència de conveniència)**



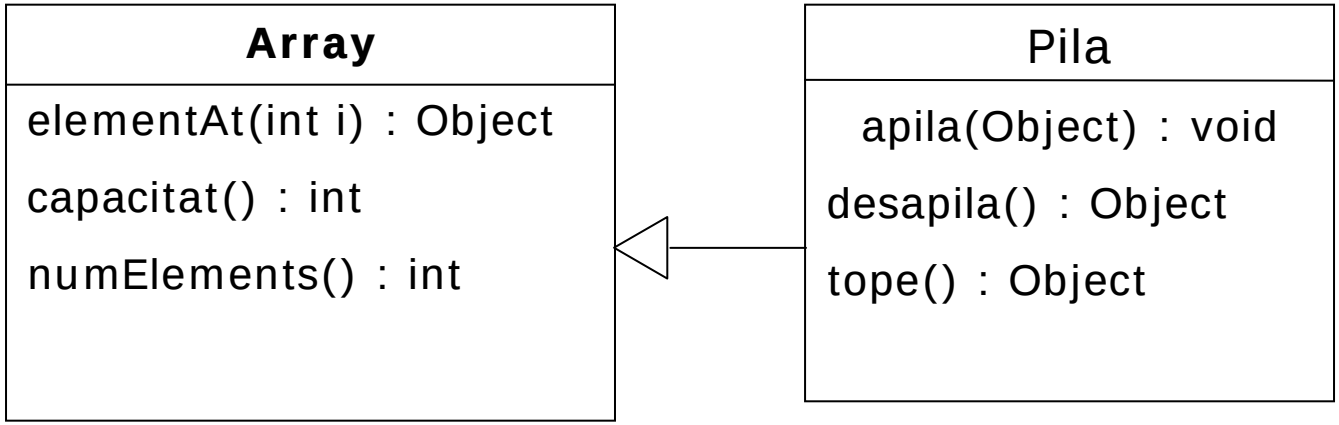
La implementació se sembla però semànticament els conceptes no estan relacionats jeràrquicament (test "és-un").

INCORRECTA!!!!

Solució: si és possible, factoritzar codi comú.
(p.ex. Ratolí i Tableta_Gràfica)

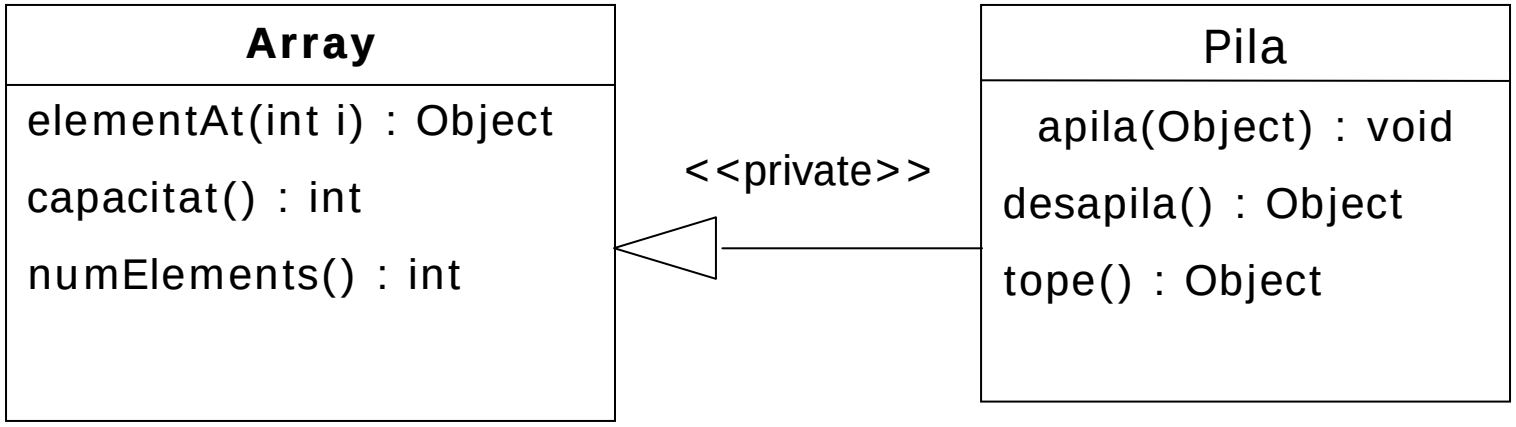
Herència de **Construcció**

- Una classe hereta part de la seua funcionalitat d'altra, modificant la interfície heretada
- La classe derivada no és una especialització de la classe base (pot ser que fins i tot no haja relació "és-un")
  - No es compleix el principi de substitució (ni es pretén)
  - P. ex., una Pila pot construir-se a partir d'un Array



Herència de **Construcció** en C++

- L'herència privada en C++ implementa un tipus d'herència de construcció que **sí** preserva el principi de substitució: també coneguda com *Herència d'Implementació Pura*
  - El fet de que Pila herete d'Array no és visible per el codi que utilitza la pila.
    - L'herència es converteix en una decisió d'implementació.



HERÈNCIA

Beneficis i costos de l'herència

Beneficis de l'Herència

- Reusabilitat software
- Compartició de codi
- Consistència d'interfície
- Construcció de components
- Prototipat ràpid
- Polimorfisme
- Ocultació d'informació

[BUDD] 8.8

Costos de l'Herència

- Velocitat d'execució
- Dimensions del programa
- Sobrecàrrega de pas de missatges
- Complexitat del programa

[BUDD] 8.9

HERÈNCIA

Elecció de tècnica de reús

Herència vs Tot-Part

- Herència és una relació entre classes, mentres que Agregació/Composició és una relació entre objectes
  - Herència es menys flexible
  - On es detecta una relació TÉ-UN no sempre es possible canviar-la per una relació d'herència. No obstant això, on es detecta una relació d'herència, **sempre** és possible reformular-la perquè es convertisca en una relació de composició.
    - Un programador de C++ és un programador
    - Tot programador de C++ té un programador al seu interior
    - Tot programador de C++ té una vocació de programar al seu interior
- **Regla del canvi:** no s'ha d'usar herència per a descriure una relació ÉS-UN si es preveu que els components puguen canviar en temps d'execució (si preveig que puga canviar la meua vocació ☺).
  - Les relacions de composició s'estableixen entre **objectes**, i per tant permeten un canvi més senzill del programa.
- **Regla del polimorfisme:** l'herència és apropiada per a descriure una relació ÉS-UN quan les entitats o els components de les estructures de dades del tipus més general poden necessitar relacionar-se amb objectes del tipus més especialitzat (e.x. per reutilització).

Elección de técnica de reuso
Introducció

- Herència (IS-A) i Composició (HAS-A) són els dos mecanismes més comuns de reús de software
  - COMPOSICIÓ (Layering): Relació tindre-un: ENTRE OBJECTES.
    - Composició significa contindre un objecte.
    - Exemple: Un cotxe té un tipus de motor.

```
class cotxe
{...
  private:
       Motor m;
};
```

  - HERÈNCIA: Relació ser-un: ENTRE CLASSES
    - Herència significa contindre una classe.
    - Exemple: Un cotxe és un vehicle

```
class cotxe: public vehicle{
  …
}
```

Elecció de tècnica de reús
Introducció

- Exemple: construcció del tipus de dada ‘Conjunt’ a partir de una classe preexistent ‘Llista’

```
class Llista{
    public:
        Llista(); //constructor
        void add (int el);
        int firstElement();
        int size();
        int includes(int el);
        void remove (int pos);
        …
};
```

- Volem que la nova classe Conjunt ens permeta afegir un valor al conjunt, determinar el nombre d’elements del conjunt i determinar si un valor específic es troba en el conjunt.

Elecció de tècnica de reús
Ús de **Composició** (Layering)

- Un objecte és una encapsulació de dades i comportament. Per tant, si utilitzem la Composició estem dient que part de l'estat de la nova estructura de dades és una instància d'una estructura existent

```
class Conjunt{
    public:
        //constructor ha d'inicialitzar l'objecte Llista
        Conjunt():lesDades(){};
        int size(){return lesDades.size();};
        int includes (int el){return lesDades.includes(el);};
        //un conjunt no pot contindre valor més d'una volta
        void add (int el){
          if (!includes(el)) lesDades.add(el);
        };

    private:
        Llista lesDades;
};
```

Elecció de tècnica de reús
Ús de **Composició** (Layering)

- La composició proporciona un mecanisme per a reutilitzar un component software existent en la creació d'una nova aplicació, simplificant la seua implementació

- La composició no realitza cap assumpció respecte a la substituibilitat. Quan es forma d'aquesta manera, un Conjunt i una Llista són tipus de dades totalment distints, i se suposa que cap d'ells pot substituir a l'altre en cap situació .

- La composició es pot aplicar de la mateixa manera en qualsevol llenguatge OO i fins i tot en llenguatges no OO.

Elecció de tècnica de reús
Ús d'**Herència**

- Amb herència una classe nova pot ser declarada com una subclasse d'una classe existent, el que provoca que totes les àrees de dades i funcions associades amb la classe original s'associen automàticament amb la nova abstracció de dades.

```
class Conjunt : public Llista{
    public:
        Conjunt() : Llista() {};
        //un conjunt no pot contindre valor més d'una volta
        void add (int el){ //refinament
          if (!includes(el)) Llista::add(el);
        };
};
```

- Implementem en termes de classe base (no objecte)
  - No existeix una llista com dada privada
- Les operacions que actuen igual en la classe base i en la derivada no han de ser redefinides (amb composició sí).

Elecció de tècnica de reús
Ús d'Herència

- L'ús de l'herència assumeix que les subclasses són a més subtipus.
  - Així, les instàncies de la nova abstracció haurien de comportar-se de manera similar a les instàncies de la classe pare.

Elecció de tècnica de reús
Composició vs. Herència

- La **composició** és una tècnica generalment **més senzilla** que l'herència.
  - Defineix més clarament la interfície que suporta el nou tipus, independentment de la interfície de l'objecte part.
  - Problema del jo-jo: En l'herència les operacions de la classe filla són un superconjunt de les de la classe pare, pel que el programador ha d'examinar totes les classes en la jerarquia per conéixer què operacions són legals per la nova estructura.
- **La composició é**s més flexible (i més resistent als canvis)
  - La composició només presuposa que el tipus de dades X s'utilitza per IMPLEMENTAR la classe C. És fàcil per tant:
    - Deixar sense implementar els mètodes que, sent rellevants per X, no ho són per a la nova classe C
    - Reimplementar C utilitzant un tipus de dades X distint sense impacte per als usuaris de la classe C.

Elecció de tècnica de reús
Composició vs. Herència

- L'**herència** presuposa el concepte de subtipus (principi de substitució)
  - L'herència permet una definició més concisa de la classe
    - Requereix menys codi.
    - Oferta més funcionalitat: qualsevol nou mètode associat al pare estarà inmediatament disponible per a tots els seus fills
  - Suporta directament el principi de substitució (composició no)
  - L'herència és lleugerament més eficient que la composició (evita una cridada).
- Desavantatges
  - Els usuaris poden manipular la nova estructura mitjançant mètodes de la classe base, fins i tot si estos no són apropiats.
  - Canviar la base d'una classe pot causar molts problemes als usuaris d'eixa classe.

Elecció de tècnica de reús
Exemples Composició vs. Herència

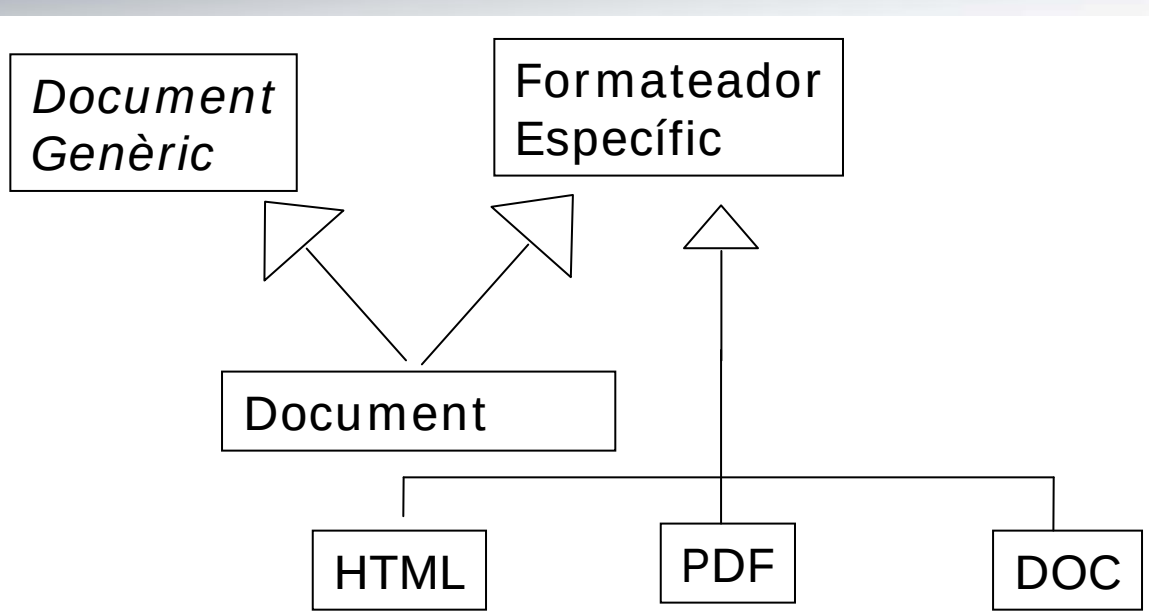
- Classe Persona i classe Empleat
  - Herència: un empleat és una persona.
- Classe Persona i classe Domicili
  - Composición: una persona tiene un domicilio
- Classe Llista i classe Node de la llista
  - Composició: una llista té un punter de tipus node al node que està al cap de la llista (tindre-un).
- Classe Empresa, classe Empleat i classe Cap de grup d'empleats
  - Herència entre empleat i cap: Un cap és un empleat
  - Composició entre empresa i empleat (o empleat i cap):
    - Una empresa pot tindre una llista d'empleats i altra de caps
    - Pel principi dels subtipus, una empresa pot tindre una única llista on apareguen tant els caps com els empleats.

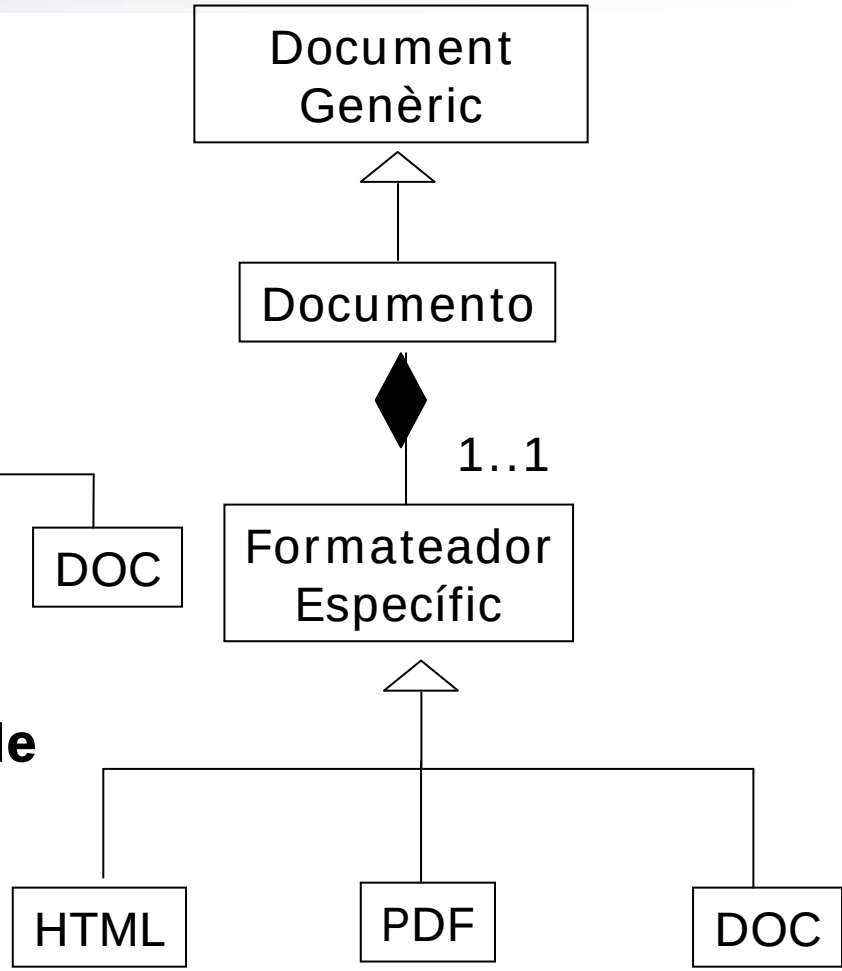
Elecció de tècnica de reús
Exercici: Visualitzador de documents

- **Suposeu** que tenim una estructura genèrica de document, i que volem visualitzar instàncies de documents amb eixa estructura en distints formats (HTML, PDF, DOC, etc.).

Com implementaríeu un sistema que suportara esta visualització?

Elecció de tècnica de reús
Solució: Visualitzador de documents





- **Amb quina de les dues solucions seria possible modificar la manera de visualització del document?**

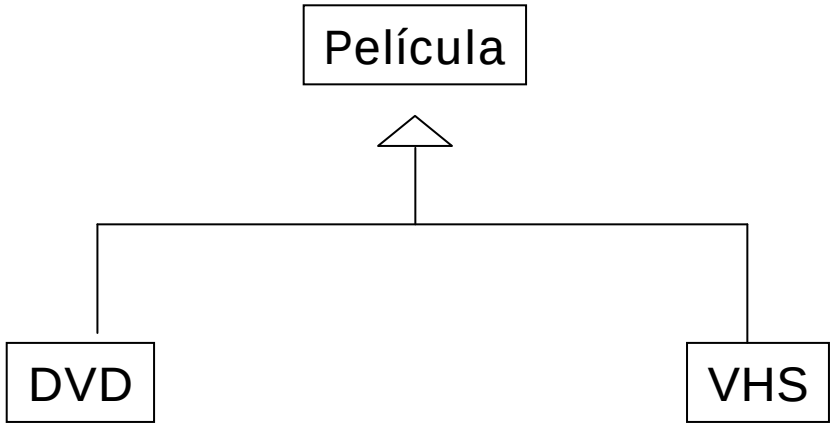
- **Amb quina es compliria el dit 'Una volta document HTML, sempre document web'?**

Elecció de tècnica de reús
Exercici: Videoclub

- **Videoclub**: Un videoclub disposa d'una sèrie de películes que poden estar en DVD o en VHS (només una cinta per película). De les películes interessa guardar el títol, l'autor, l'any d'edició i el idioma (o els idiomes, en cas de DVD). El preu d'alquiler de les películes varia en funció del tipus de película.
- **Què classes implementaríeu i com les relacionaríeu?**

Elecció de tècnica de reús
Solució: Videoclub

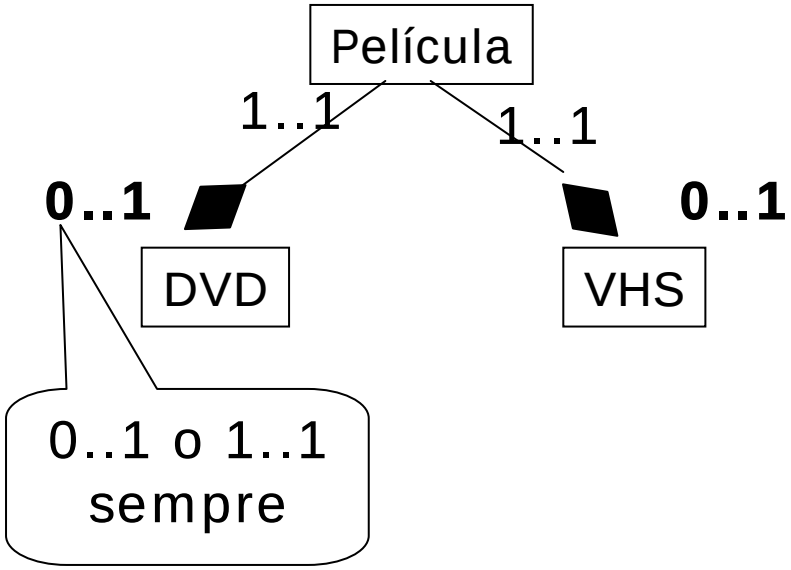
- Mitjançant herència:



- Canviar de VHS a DVD suposaria eliminar la instància de película i tornar-la a crear (l'herència és una *relació inmutable entre classes*). On treballava amb instància de VHS (e.x. visualitzar) ara he de treballar amb instàncies de DVD.

Elecció de tècnica de reús
Solució: Videoclub

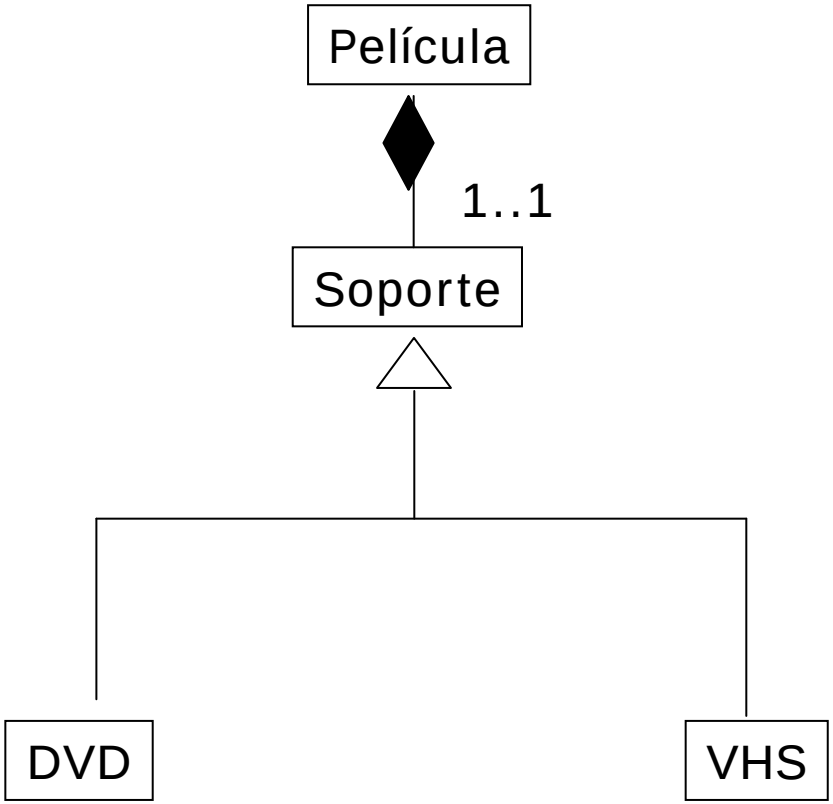
- Mitjançant composició:



- Més flexible, ja que composició implica una relació a nivell d'objecte (no de classe).
- Com indique que he de tindre un DVD o un VHS?

Elecció de tècnica de reús
Solució: Videoclub

- Mitjançant composició i herència:

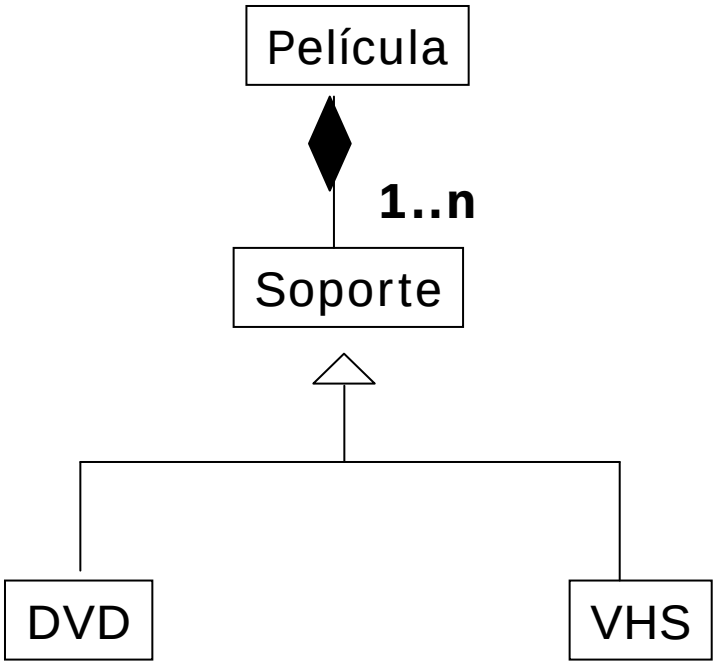


- Ara si jo renove el meu parc de películes i canvie les VHS per DVD no hi ha cap problema, perquè puc treballar amb instàncies de soporte (e.x. visualitzar), i ambdos són subtipus de soporte. S'aprofiten les característiques de l'herència, i s'afegeix flexibilitat.

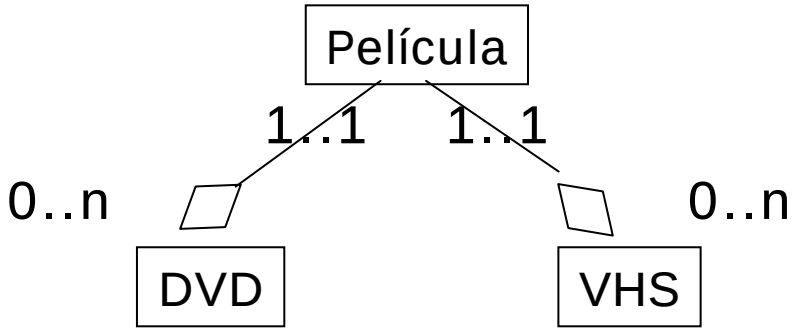
Elecció de tècnica de reús
Exercici: Videoclub amb vàries còpies

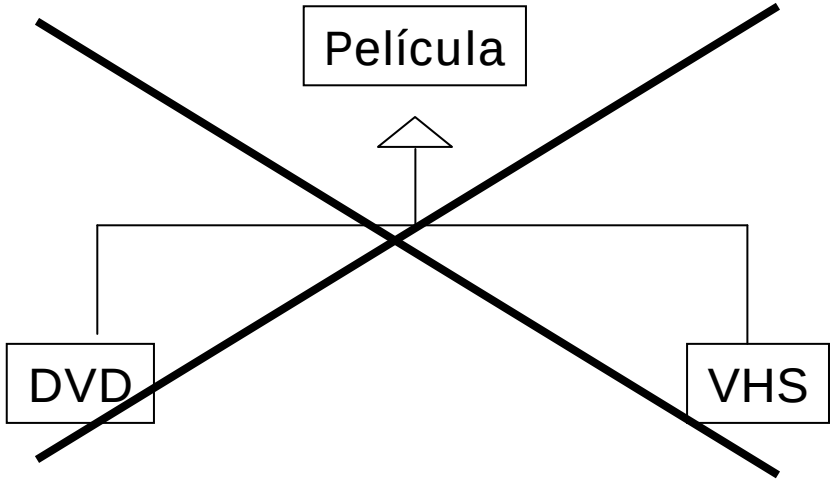
- Què canvis hauria de realitzar en una i altra solució si el creixement del meu videoclub aconsellara tindre vàries còpies d'una mateixa película?

Elecció de tècnica de reús
Solució: Videoclub amb vàries còpies



Agregació en lloc de composició





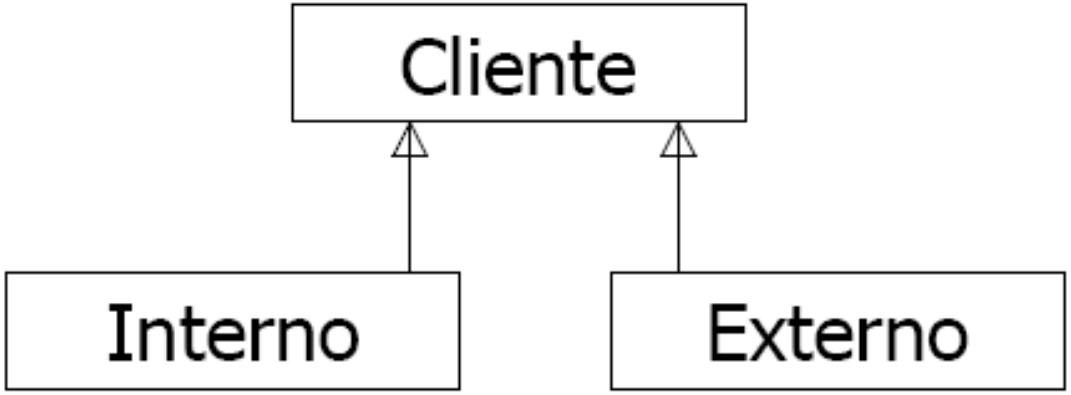
HERÈNCIA

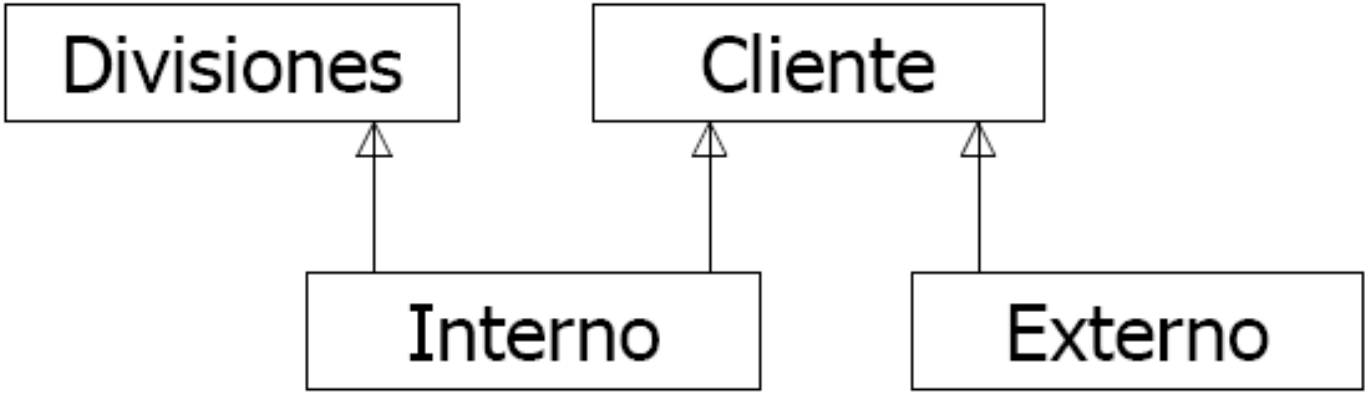
EXERCISIS

Relacions entre Classes i Objectes
Exercici 1

- Dibuixeu un diagrama de classes que mostre les dues categories que existeixen de **Clients** d'una empresa: **Clients externs**, que són oltres companyies, i **Clients interns**, que són totes les divisions de dintre de l'empresa.
- Suposeu ara que **Divisió** (les instàncies de la qual són les distintes divisions de l'empresa) és una classe del sistema. Canviaria en alguna cosa la vostra solució?

Relacions entre Classes i Objectes
Exercici 1: Solució

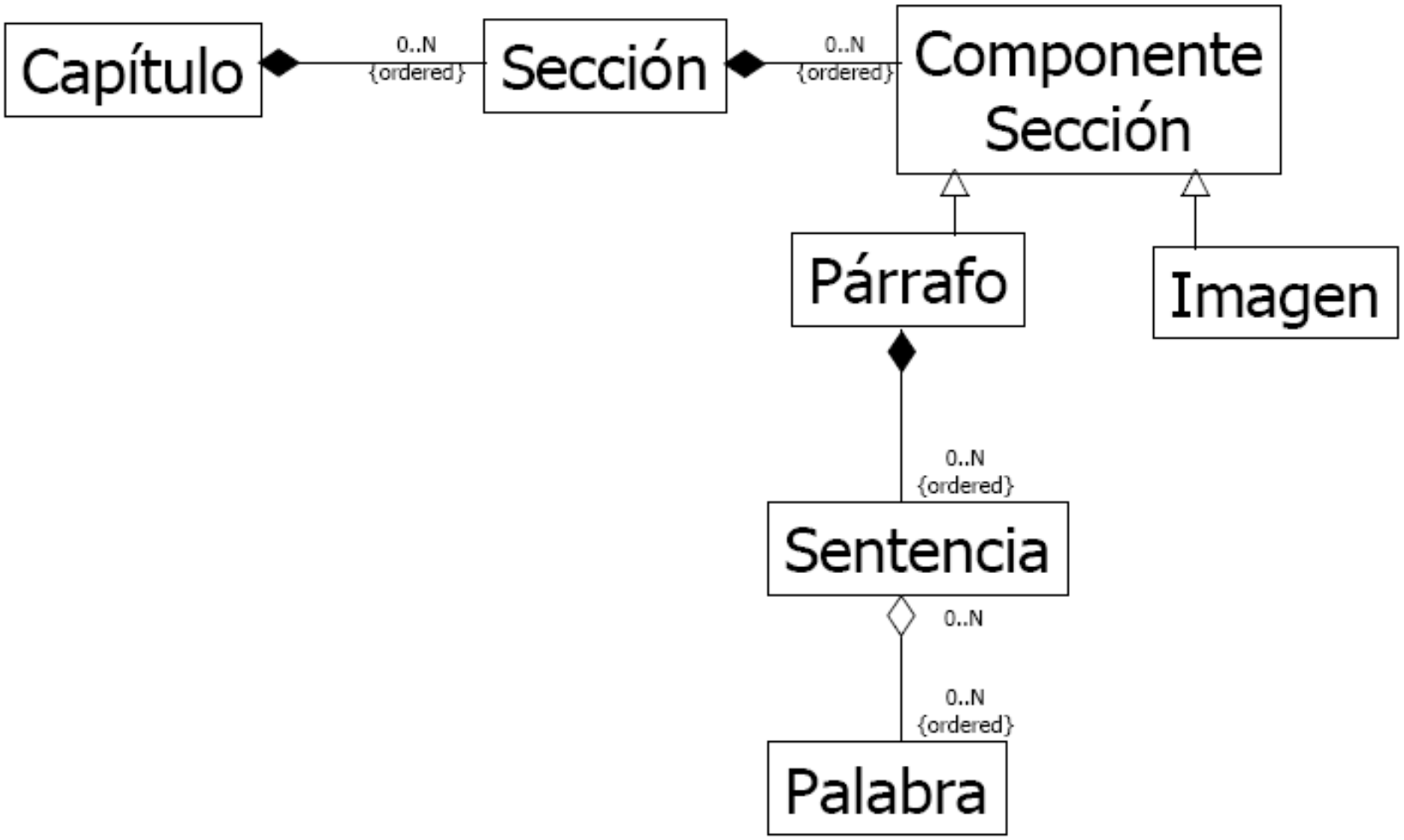




Relacions entre Classes i Objectes
Exercici 2

- Dibuixeu un diagrama de classes que mostre l'estructura d'un capítol de llibre; un capítol està compost per vàries seccions, cadascuna de les quals comprén diversos paràgrafs i figures. Un paràgraf inclou vàries sentències, cadascuna de les quals conté vàries paraules.
- Suposeu que en un futur es preveu que el sistema gestione a més de paràgrafs i figures altres components, com taules, llistes, vinyetes, etc.
- Suposeu a més que una paraula pot aparéixer en vàries sentències.

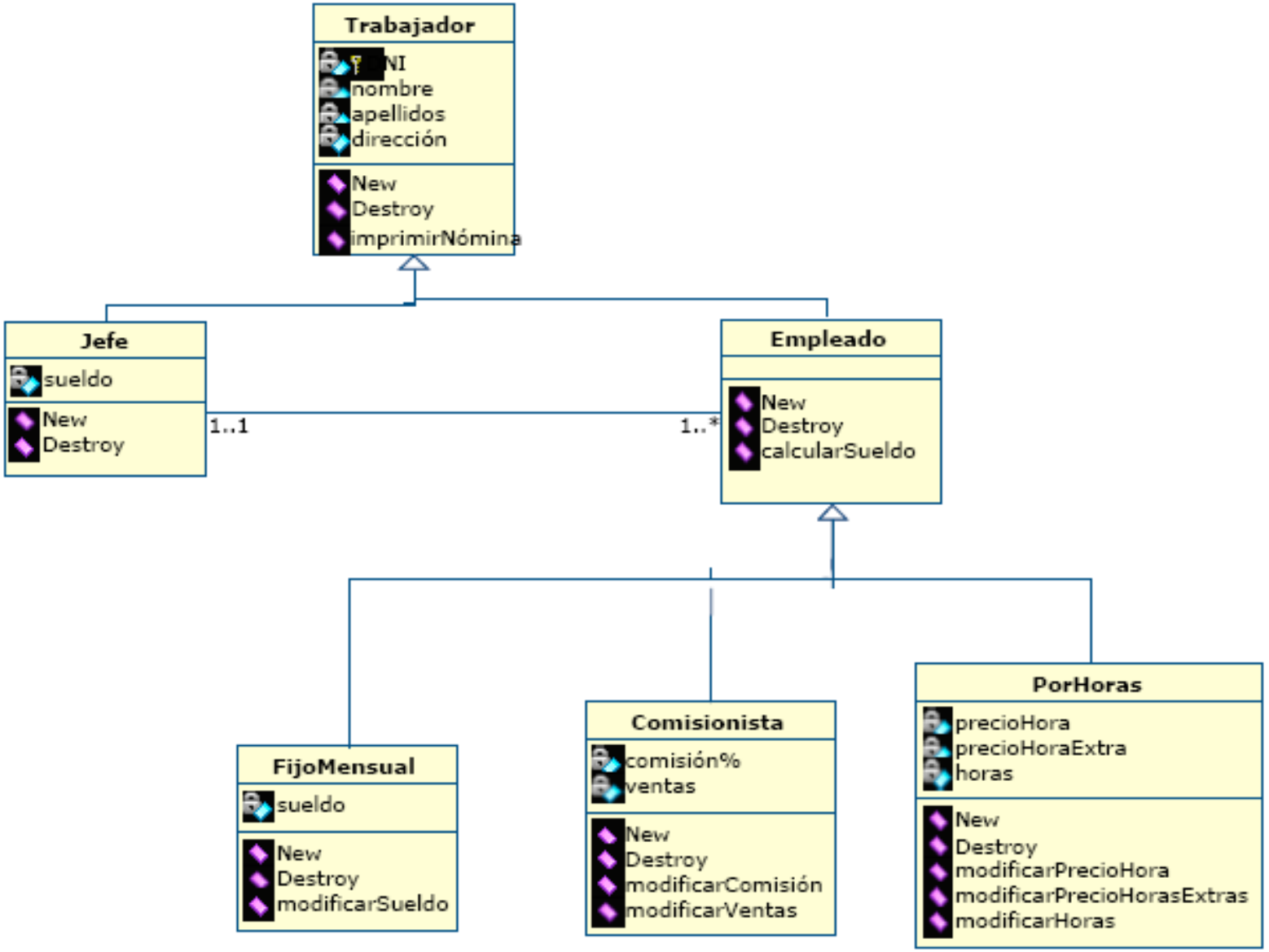
Relacions entre Classes i Objectes
Exercici 2: Solució



Relacions entre Classes i Objectes
Exercici 3

- Es desitja desenvolupar un sistema de nòmina per als treballadors d'una empresa. Les dades personals dels treballadors són Nomb i Cognoms, Direcció, DNI. Existeixen diferents tipus de treballadors:
  - Fixos Mensuals: que cobren una quantitat fixa al mes
  - Comisionistes: cobren un percentatge fixe per les ventes que han realitzat
  - Per Hores: cobren un preu per cada una de les horaes que han realitzat durant el mes. El preu és fixe per a les primeres 40 hores i és altre per a les hores realitzades a partir de la 40 hora mensual.
  - Cap: cobra un sueldo fixe (no s'ha de calcular)
- Cada empleat té obligatòriament un cap (exceptuant els caps que no tenen cap). El programa ha de permetre donar de alta treballadors, fixar els seus emoluments, hores o ventes realitzades e imprimir la nòmina corresponent al final de mes.

Relaciones entre Clases y Objetos
Ejercicio 3: Solución



Resum

- Una de les claus del poder de la programació orientada a objectes és assolir la reutilització del software a través de l'herència.
- El programador pot designar que la nova classe herete les dades membre i funcions membre d'una classe base definida prèviament. En aquest cas, la classe nova es coneix com classe derivada.
- En l'herència simple, una classe es deriva a partir d'una sola classe base. En l'herència múltiple, una classe derivada hereta de diverses classes base (que possiblement no tinguen relació entre elles).
- Una classe derivada en general agrega les seues pròpies dades membre i funcions membre, pel que comunament té una definició major que la seua classe base. Una classe derivada és més específica que la seua classe base i en general representa menys objectes.
- Una classe derivada no pot accedir a les dades membre private de la seua classe base; permetre això violaria l'encapsulament de la classe base. No obstant això, una classe derivada pot accedir a les dades membre public i protected de la seua classe base.
- Un constructor de classe derivada sempre crida primer al constructor de la seua classe base, perquè cree i inicialitze els membres de classe base de la classe derivada.

Resum

- Els destructors s'invoquen en l'ordre invers al de les cridades de constructor, pel que el destructor de la classe derivada s'invoca abans que el destructor de la seua classe base.
- L'herència permet la reutilització del software, la qual estalvia temps en el desenvolupament i promou l'ocupació de software d'alta qualitat prèviament provat i depurat.
- L'herència es pot fer a partir de biblioteques de classes ja existents.
- Algun dia, la major part del software es construirà a partir de components reutilitzables estandarditzats, de la mateixa manera que es construeix la major part del hardware.
- L'implementador d'una classe derivada no necessita accedir al codi font de la seua classe base, però si necessita interactuar amb la classe base i el codi objecte de la classe base.
- Un objecte d'una classe derivada pot ser tractat com objecte de la seua classe base pública corresponent. No obstant això, el contrari no és cert.
- Una classe base existeix en una relació jeràrquica amb les seues classes derivades simples.

Resum

- Una classe pot existir per si mateixa. Quan s'utilitza aquesta classe amb el mecanisme d'herència, es torna una classe base que subministra atributs i comportaments a altres classes o es torna una classe derivada que hereta eixos atributs i comportaments.
- Les jerarquies d'herència poden tenir una profunditat arbitrària dins de les limitacions físiques del seu sistema particular.
- Les jerarquies són eines útils per a entendre i administrar la complexitat. Ara que el software es torna cada vegada més complex, C++ ofereix mecanismes per a manejar estructures jeràrquiques mitjançant herència i polimorfisme.
- És possible utilitzar una conversió mitjançant cast explícita per a convertir un apuntador de classe base en apuntador de classe derivada. Tal apuntador no s'ha de desreferenciar, llevat que en realitat apunte a un objecte del tipus de la classe derivada.
- L'accés protected serveix com nivell intermedi de protecció entre l'accés public i l'accés private. Els membres i friend de la classe base i els membres i friend de les classes derivades poden accedir els membres protected d'una classe base; cap altra funció pot accedir als membres protected d'una classe base.

Resum

- Els membres protected s'utilitzen per a estendre els privilegis a les classes derivades, negant aquests privilegis a les funcions que no són de la classe ni són friend.
- L'herència múltiple s'indica posant dos punts (:) a continuació del nom de la classe derivada i després una llista separada per comes de classes base. Per a cridar als constructors de la classe base, s'utilitza sintaxi d'inicialitzador de membres en el constructor de la classe derivada.
- Quan es deriva una classe a partir d'una classe base, aquesta es pot declarar com public, protected o private.
- Quan es deriva una classe a partir d'una classe base public, els membres public d'aquesta es tornen membres public de la classe derivada i els membres protected de la classe base es tornen membres protected de la classe derivada.
- Quan es deriva una classe a partir d'una classe base protected, els membres public i protected d'aquesta es tornen membres protected de la classe derivada.
- Quan es deriva una classe a partir d'una classe base private, els membres public i protected d'aquesta es tornen membres private de la classe derivada.

Resum

- Una classe base pot ser una classe base directa o indirecta d'una classe derivada. Una classe base directa és aquella que es llista explícitament quan es declara la classe derivada. Una classe base indirecta no es llista explícitament; en canvi, s'hereta de diversos nivells més amunt en l'arbre jeràrquic.
- Quan un membre de la classe base és inadequat per a una classe derivada, simplement es redefineix dit membre en la classe derivada.
- És important distingir entre les relacions "és un" i "té un". En les relacions "té un", un objecte de classe té com membre un objecte d'altra classe. En les relacions "és un", un objecte d'un tipus de classe derivada també pot ser tractat com objecte del tipus de la classe base. "És un" significa herència. "Té un" significa composició.
- És possible assignar un objecte de classe derivada a un objecte de classe base. Aquest tipus d'assignació té sentit, doncs la classe derivada té membres que corresponen a cadascun dels membres de la classe base.
- És possible convertir implícitament un apuntador a un objecte de classe derivada en apuntador a un objecte de classe base.

Resum

- És possible convertir un apuntador de classe base en apuntador de classe derivada mitjançant una conversió cast explícita. La destinació ha de ser un objecte de classe derivada.
- Una classe base especifica comunitat. Totes les classes derivades d'una classe base hereten les capacitats d'aquesta. En el procés de disseny orientat a objectes, el dissenyador cerca la comunitat i la factoritza per a formar classes base desitjables. Després les classes derivades es personalitzen més enllà de les capacitats heretades de la classe base.
- La lectura d'un conjunt de declaracions de classe derivada pot ser confusa, doncs no tots els membres de la classe derivada estan presents en aquestes declaracions. En particular, els membres heretats no es llisten en les declaracions de les classes derivades, però aquests membres de fet estan presents en les classes derivades.
- Les relacions "té un" són exemples de la creació de classes noves per mitjà de composició de classes ja existents.
- Les relacions "coneix un" són exemples d'objectes que contenen apuntadors o referències a altres objectes, pel que estan conscients d'ells.

Resum

- Els constructors d'objectes membre s'invoquen en l'ordre que es declaren els objectes. En l'herència, els constructors de classe base s'invoquen en l'ordre que s'especifica l'herència i abans del constructor de la classe derivada.
- En el cas d'un objecte de classe derivada, primer s'invoca al constructor de la classe base i després al de la classe derivada (que pot cridar a constructors d'objectes membre).
- Quan es destrueix l'objecte de la classe derivada, s'invoquen els destructors en l'ordre invers al dels constructors: primer es diu al destructor de la classe derivada, després al de la classe base.
- És possible derivar una classe a partir de més d'una classe base; tal derivació es diu herència múltiple.
- L'herència múltiple s'indica posant a continuació del caràcter de dos punts (:), que indica herència, una llista separada per comas de classes base.
- El constructor de la classe derivada crida als constructors de les diferents classes base per mitjà de la sintaxi de inicialitzador de membres. Els constructors de classes base s'invoquen en l'ordre que es declaren les classes base durant l'herència.